TURING 图灵程序设计丛书

Interactive Data Visualization for the Web

数据可视化实战

使用D3设计交互式图表

[美] *Scott Murray* 著
李松峰 译

O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明


图灵程序设计丛书

数据可视化实战
使用D3设计交互式图表

Interactive Data Visualization for the Web

[美] Scott Murray 著
李松峰 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　　京

图书在版编目（C I P）数据

数据可视化实战 : 使用D3设计交互式图表 / (美)
莫瑞 (Murray,S.) 著 ; 李松峰译. -- 北京 : 人民邮电
出版社, 2013.6
（图灵程序设计丛书）
书名原文: Interactive data visualization for
the Web
ISBN 978-7-115-32011-7

Ⅰ. ①数… Ⅱ. ①莫… ②李… Ⅲ. ①可视化软件
Ⅳ. ①TP31

中国版本图书馆CIP数据核字(2013)第115661号

内 容 提 要

数据可视化是展示数据的重要手段，广泛适用于数据分析、计量统计、演讲展示和各种网站应用。而通过浏览器来呈现数据不受平台限制，任何计算机只要能上网就可以看到漂亮的交互式图表。本书将带领读者学习当前最热门的基于浏览器的数据可视化库——D3。作者通过风趣幽默的语言、简单易懂的示例，由浅入深地介绍了使用D3所需的基本技术，以及基于数据绘图、比例尺、数轴、数据更新、过渡和动画等构建交互式在线图表的核心概念，最后还介绍了D3中常用的布局方法和创建地图等流行应用的技巧。

本书需要读者具有一定的Web开发经验，特别要了解一些DOM编程。除此之外，只要对数据可视化感兴趣，均可阅读学习。

◆ 著 [美] Scott Murray
译 李松峰
责任编辑 刘美英
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：787×1092 1/16
印张：15.5
字数：295千字 2013年6月第1版
印数：1－3 000册 2013年6月北京第1次印刷
著作权合同登记号 图字：01-2013-3660号

定价：59.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创 新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

前言

本书是关于数据可视化的，但非专业程序员也可以看懂。如果你是一位艺术家或者拥有视觉表现经验的图形设计师，那么这本书就是为你写的。如果你是一位专栏作者或者研究人员，但之前没有可视化或编程经验，那这本书也是写给你的。

本书介绍 JavaScript 的数据可视化库 D3（http://d3js.org/），它可以把数据加载到网页中并基于数据生成各种图表。要看懂这本书，之前有没有编程经验不太重要。也许你以前写过程序，也听说过关于 JavaScript 语言的各种传闻，那你可以从 D3 和数据可视化入手，跟 JavaScript 第一次亲密接触。没错，JavaScript 是有那么一点点古怪，但并没有你听说得那么坏，一切其实都很好。请坐，稍安毋躁。

本书脱胎于我在自己网站上发布的一系列文章。当时（2012 年 1 月），还很难找到面向新手的 D3 学习资料。我的网站访问量很快就达到每天几百，甚至几千次，这说明人们对这个领域（尤其是 D3）的关注度与日俱增。如果你看过那一系列教程，那对本书内容会很熟悉。不过，我也补充了很多新内容，包括更多的示例、有用的提示以及建议。此外，本书 78% 以上都是冷笑话。

数据可视化是一个跨学科的领域，因此一本书不可能涵盖所有技术。好在，随着这个领域越来越热门，市面上也有很多这类书可以选择，能够起到相互补充的作用。

比如，有讨论设计流程的：

- *Designing Data Visualizations: Intentional Communication from Data to Display*，作者是 Noah Iliinsky 和 Julie Steele（O’Reilly Media，2011）；
- *Data Visualization: A Successful Design Process*，作者 Andy Kirk（Packt Publishing，2012）。

有关于视觉设计原理和技术的：

- *The Functional Art: An Introduction to Information Graphics and Visualization*，作者 Alberto Cairo（New Riders，2012）；
- *Information Dashboard Design: The Effective Visual Communication of Data*，作者 Stephen Few（O'Reilly Media，2006）。

还有探讨数据实战的：

- *Bad Data Handbook: Mapping the World of Data Problems*，作者 Q. Ethan McCallum（O'Reilly Media，2012）；
- *Data Analysis with Open Source Tools: A Hands-On Guide for Programmers and Data Scientists*，作者 Philipp K. Janert（O'Reilly Media，2010）；
- *Python for Data Analysis: Agile Tools for Real World Data*，作者 Wes McKinney（O'Reilly Media，2012）。

排版约定

本书使用的排版约定如下。

- 楷体
  表示新的术语。

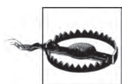
- 等宽字体
  表示程序片段，也用于在正文中表示程序中使用的变量、函数名、命令行代码、环境变量、语句和关键词等代码文本。

- 加粗的等宽字体
  表示应该由用户逐字输入的命令或者其他文本。

- 倾斜的等宽字体
  表示应该由用户输入的值或根据上下文决定的值替换的文本。



这个图标代表小窍门、建议或说明。



这个图标代表警告信息。

使用代码

本书就是要帮读者解决实际问题的。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。因此，用本书中的几段代码写成一个程序不用向我们申请许可。但是销售或者分发 O'Reilly 图书随附的代码光盘则必须事先获得授权。引用书中的代码来回答问题也无需我们授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版商和 ISBN，例如：*Interactive Data Visualization for the Web* by Scott Murray (O'Reilly). Copyright 2013 Scott Murray, 978-1-449-33973-9。

如果还有其他使用代码的情形需要与我们沟通，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online



Safari Books Online（www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/interactive_data_visualization_web

中文版地址：

http://www.oreilly.com.cn/index.php?func=book&isbn=978-7-115-32011-7

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

我的名字虽然印在了封面上，但作为作者，我感觉自己只不过是一个“漏斗”。本书每一页的内容其实是萃取了几百位杰出人物智慧的结晶。

首先，我必须感谢我妻子。要不是她提醒我“嘿，你应该把那些教程整理成一本书”，要不是她在背后支持和鼓励我，这本书不可能面世。

感谢 Rosten Woo，我的第一个 D3 项目就是跟他一起做的，是他带我结识了这个新工具，最后欲罢不能。感谢 Joe Golike 回应我们早期关于 D3 调试的问题。还要感谢 Jen Lowe 和 Sha Hwang 审校原来的教程和给出的意见。

非常感谢 Casey Reas、Dan Shiffman、Joshua Noble 和 Noah Iliinsky，不仅仅因为他们对本书提出了宝贵建议，更重要的是感谢他们在艺术、设计、编码和数据领域突破性的工作。他们的职业道路对我影响深远。

同样，我还要感谢 MassArt Dynamic Media Institute 的 Jan Kubasiewicz。2007 年，Jan 鼓励我接触了一个叫 Processing 的东西。从那时起，我的职业生涯就完全转向了编程艺术设计、数据可视化。今天，又有了这本书。

我跟编辑 Meghan Blanchette 及 O’Reilly 的其他人合作非常愉快。感谢 Meghan 和她的团队为出版这本书前前后后地忙碌了那么多天，让本来无法触及的思想变成了现实当中看得见摸得着的一本书，而且还有很多字和奇形怪状的图表印在了里面。

特别感谢 Mike Bostock、Jen Lowe、Anna Powell-Smith 和 Daisy Vincent 答应做本书的技术审校，并给出了很多特别有价值的反馈。最终内容质量大幅提升，主要源于他们的反馈。换句话说，要是你在代码示例中发现了错误，那肯定是因为他们要求我修改，而我坚持没改造成的。

Mike 当然是最应该感谢的人了，他开发了 D3。如果没有这个精美的程序，数据可视化社区就不会像今天这样充满热情、活力四射，对标准的遵行也不会那么到位。

说到社区，还要感谢 Jérôme Cukier、Lynn Cherny、Jason Davies、Jeff Heer、Santiago Ortiz、Kim Rees、Moritz Stefaner、Jan Willem Tulp，还有其他没提到的 D3 邮件列表中的人，以及我身边对我的思考和写作给予了直接和间接帮助的人。谢谢你们的支持。我感到非常荣幸，能与那么多天才的人们交流学习。

第 1 章

写在前面

1.1 数据为什么要可视化

这个信息时代更多地让人觉得它是个信息过剩的时代。铺天盖地般的信息令人目不暇接，很多未经加工的原始信息只有使用某种方法找出其中的规律才有价值。

谢天谢地，我们人类是对图形图像极为敏感的生物。虽然很少有人能从一堆数字中发现趋势，但即使是小孩子也能看懂条形图，并且能从这些图形中明白数字的含义。正因为如此，数据可视化成了一股潮流。可视化数据成为与人沟通的最便捷方式。

当然，数据可视化跟用语言描述一样，都可能“撒谎”、误导人，甚至扭曲事实。不过，只要潜心学习，多加小心，把数据变成生动的图表就能帮我们从一个全新的角度来看懂这个世界，从中揭示出原先隐藏的一些模式和趋势。运用得当，数据可视化是可以开口讲故事的。

如果从字面上来理解，可视化就是把信息映射为可见图形的过程。我们必须总结出一些规则，解读数据，同时把数据变成有形的东西。比如图 1-1 中这个最基本的条形图吧，它就是根据一个最简单的规则生成的：较大的数值映射为较高的条形。

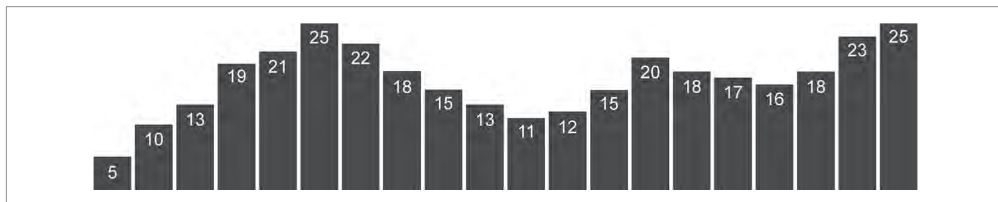


图 1-1：将数据值映射为条形 [1]

注 1：本书部分彩图请在图灵社区本书页面下载：http://www.ituring.com.cn/book/1126。——编者注

图形越复杂，数据集越复杂，规则也就越复杂。

1.2 为什么要写代码

手工完成数据到图形的映射不是不行，但效率太低，也太乏味。所以，一般我们都要利用计算机来提高工作效率。效率上来了，才能把时间放在研究更大的数据集上，这样才能处理几千条数据，甚至几百万条数据。这么大的数据量，如果全靠手工得几年，用计算机不过瞬间而已。还有一点同样重要，利用计算机可以快速验证不同的映射思路，改一改规则然后就能即刻看到重新生成的输出结果。这个不断重复的“写代码 – 呈现 – 评估”的过程，是快速迭代、设计出最优映射规则的关键。

映射规则发挥的是设计系统的作用，不用人手工描绘，计算机帮我们干。我们人呢，应该把精力放在挖概念、找联系和写规则上面，其他统统让计算机帮我们搞定。

遗憾的是，计算机软件（通常是计算模型）很难精确表达人类的所思所想。（说句公道话，很多人其实也不善于表达自己的想法。）因为计算机是二进制系统，所有一切都是开关、是否、这个那个、这里或非这里。人类作为感性宽厚的生物，在计算机不愿意迁就我们的时候，我们只能迁就它。于是不可避免地，我们就要学习写代码，编软件。我们学着跟计算机沟通，使用非常有限但又很精确的语法，好让计算机能够更好地理解我们。

看见自己做出来的可视化效果那么棒，我们的心都醉了，于是就会继续写代码。而一旦目睹前所未见的可视化效果，我们又欲罢不能，于是就不断重复着这个过程。恰似像神秘的数据魔瓶里跑出来一个天才的视觉魔法师。

1.3 为什么要交互

静态可视化展示的只是预先合成好的数据“视图”，而要展示相同信息的不同侧面，往往需要多个静态视图。在静态视图中，数据的维度同样也是受限的，因为所有可视化的要素必须同时展示在同一个表面上。要在静态视图中表现多维数据的难度势比登天。固定不变的图像什么时候最合适？除非像印刷或打印时，不需要也没有必要弄一堆视图。

然而，动态的响应式的图形可以激发人们探索数据的欲望。1996 年，马里兰大学的 Ben Shneiderman 率先在“Visual Information-Seeking Mantra”中提出“先给出一个大小合适、筛选得当的概要，然后根据需要展示细节”，由此开始，大多数交互可视化工具的基本功能就发生了变化。

今天，许多交互式可视化作品中都有这种设计模式的影子。不同功能的组合是有效的，因为无论你只想大概浏览或了解一下数据集，还是带着某个疑问想从可视化图形中找到答案，这种模式都可以胜任。总之，展示数据概要同时又配有一系列“挖掘”工具的交互式可视化作品，能够同时满足多种用户的需要，无论他是相关领域的新人，还是已经非常熟稔于相关数据。

当然啦，交互性也能起到静态图像没办法起到的鼓励参与的作用。动态切换和制作精美的界面，经常会让探索数据的人产生玩游戏的感觉。因此，交互可视化能够把那些原本会对相关主题和数据视而不见的人吸引过来，你说它重要不重要？

1.4　为什么要在Web上

看不见的可视化不叫可视化。生成了可视化作品，把它展示给别人至关重要，而在 Web 上发布是向全世界展示的最快捷方式。采用标准的 Web 技术，意味着只要是使用较新浏览器的人都能看到和体验你的成果，而跟他们使用的操作系统（Windows、Mac、Linux）和设备（笔记本、台式机、智能手机、平板电脑）无关。

最关键的是，本书介绍的一切都可以通过免费工具来实现。因此，作为学习者，你唯一的成本就是自己的时间。本书讨论的一切都基于开源的、符合 Web 标准的技术。

避开了专有软件和插件，就能保证你的工作成果可以无障碍地送达各种设备，台式计算机、平板电脑、智能手机，都没问题。作品越容易被人看到，你的受众就越多，你的影响就越大。

1.5　这是一本什么书

本书基于 D3 这个强大的 Web 可视化展示工具，涉及数据可视化、交互设计和 Web 开发这三个主题。全书所有内容都是我在学习使用 D3 期间摸索和积累的经验教训的结晶。可能很多读者（包括我自己）原来都有一些设计、绘图和数据可视化的经验，但也许对编程和计算机知道的不多。

D3 被误解说很难学，我觉得这是不公允的。D3 没有那么复杂，而是它立足的 Web 有点复杂。不过，要想熟练地使用 D3，的确需要一些先行的 Web 知识，包括 HTML、CSS、JavaScript 和 SVG。很多人（包括我自己）都是自学的这些技术。自学本身不是问题，因为门槛很低，但它存在一定问题，因为你不一定会从头学起，所以有时候难免会采用一些偏门手法应付差事，得过且过。说实话，真要用好 D3，

就得老老实实地掌握一些基础知识。

因为 D3 是用 JavaScript 写出来的，所以学习它通常意味着要理解很多 JavaScript 代码。对我们很多搞数据的人而言，D3 是他们学习 JavaScript（乃至通常所说 Web 开发）的起点。单学一门语言已经够难的了，更何况是用这门语言写的新程序库呢。但掌握了 JavaScript，你就可以利用 D3 尝试一些从未做过的事。难学归难学，我敢保证你在这门语言和新工具上投入的时间会得到意想不到的回报。

我写这本书的目的就是缩短读者的学习时间，好让你尽早开始着手做出好东西来。本书会采取由浅入深的思路，先从基本概念讲起，慢慢地过渡到复杂主题。我不想过多地跟大家介绍具体的可视化效果，而主要把篇幅集中于全面深入地探讨 D3 的工作原理，以便你将来可以随心所欲地利用它来生成符合自己需要的作品。

1.6 读者是谁

你完全可以是一位新手，不用了解数据可视化，不用了解 Web 开发，甚至两都不需要了解。（放心吧！）也许你是一位专栏作者，想把采访过程中收集的数据以可视化形式展示出来。也许你是一位设计师，绘制一幅静态的信息图对你来说只是信手拈来的事，但你希望能更上一层楼，掌握制作 Web 交互作品的技术。也许你是一位艺术家，醉心于根据数据来生成艺术作品。也许你是一位程序员，对 JavaScript 和 Web 并不陌生，但强烈希望掌握一个新工具，体验一番数据可视化的乐趣。

好了，不管你是谁，我都希望你：

- 听过说“万维网”，或者知道什么叫“上网”；
- 稍微懂点 HTML、DOM 和 CSS；
- 甚至有点编程经验；
- 听说过 jQuery，或者写过 JavaScript 代码；
- 听到 CSV、SVG 或 JSON 这些新词汇，不会被吓懵；
- 想做一些有价值的交互式可视化项目。

即使上面提到的这些东西你没听过说或不太了解，也不用担心。只要花点时间看看第 3 章就好了，那一章介绍了在学习 D3 之前真正需要了解的东西。

1.7 这不是什么书

这不是一本计算机专业人士才能看懂的书，不是一本教人深入学习错综复杂的 Web

技术的书。

本着这个原则，我可能会在书中隐藏一些技术细节，粗略地讲解一些重要的基础概念。至于讲解的方式，恐怕会让资深软件工程师不屑一顾。这没什么，因为这本书是写给艺术家和设计师的，不是写给工程师大牛的。我们只会涉及基本概念，要想深入某些技术细节，你要量力而行。

另外，我有意没有给出某些问题的全部解决方案，而是只向大家推荐我认为最简单的方案，就算不是最简单，也应该是最好理解的。

我的目标是让大家理解 D3 的基本概念和方法。为此，本书就没有办法围绕特定的示例项目来组织。而每个人的设计目标和数据集千差万别，所以只要掌握了基本概念和方法，至于怎么用、用到哪，就完全悉听尊便了。

1.8 使用示例代码

如果你是一位天才，可能不用看示例文件就能学会 D3。如果是这样，本节下面的内容可以不看。

如果你跟我一样，虽然很聪明，但还没有达到天才的地步，那恐怕必须借助本书随附的示例代码，才能看懂这本书。在看书之前，请先到 GitHub 上下载完整的示例代码文件（http://t.cn/zTI9BXG）。[1]

正常人通常是点击 ZIP 文件的链接来下载，而骨灰级玩家愿意使用 Git 来克隆代码。如果你听不太明白后半句话是什么意思，就按照前半句话的意思做好了 。

在下载到的压缩文件中，每章的代码文件都放在以相应章名命名的文件夹里，比如：

```
chapter_04
chapter_05
chapter_06
chapter_07
chapter_08
...
```

文件按章组织，因此第 9 章在提到 01_bar_chart.html 时，你就知道这个文件的位置是：d3-book/chapter_9/01_bar_chart.html。

只要不是出于商业目的，你可以随便复制、使用、修改和重用这些代码。

注 1：也可以从图灵社区下载：http://www.ituring.com.cn/book/1126。——编者注

1.9 谢谢你

最后我想说，我已经尽最大努力把这本书写好，而且为了保证大家的学习效果反复进行了修改。谢谢你看这本书，希望你能从中学习到知识，甚至感受到乐趣。

第 2 章 D3简介

D3（有时候也叫 D^3 或 d3.js）是一个 JavaScript 库，用于创建数据可视化图形。但这么说多少还是有点低估它了。

事实上，D3 是一个缩写，它的全称叫 Data-Driven Documents（数据驱动的文档）。数据来源于你，而文档就是基于 Web 的文档（或者网页），代表可以在浏览器中展现的一切，比如 HTML、SVG。D3 扮演的是一个驱动程序的角色，因为它联系着数据和文档。

当然，这个名字也能让人直观地联想到 Web 开发背后的那个关键词：W3（即 World Wide Web，万维网），现在简称“Web”（也就是人们常说的“上网”的“网”）。

D3 的主要作者是才华横溢的 Mike Bostock，此外还有几位贡献者。这个项目完全是开源的，托管在 GitHub 上（https://github.com/mbostock/d3/），任何人都可以自由使用。

D3 的许可方式是 BSD，因此无论你出于商业还是非商业目的使用、修改和整合它，都不用付出任何代价。

D3 官方网站是 http://d3js.org。

2.1 D3能做什么

简单地说，D3 是一个很不错的软件，它能帮你生成和操作带数据的文档。为此，要

经历以下几步：

- 把数据加载到浏览器的内存空间；
- 把数据绑定到文档中的元素，根据需要创建新元素；
- 解析每个元素的范围资料（bound datum）并为其设置相应的可视化属性，实现元素的变换（transforming）；
- 响应用户输入实现元素状态的过渡（transitioning）。

学习 D3 的过程，就是学习那些告诉它如何加载、绑定数据，变换和过渡元素的语法的过程。

其中，变换这一步最重要，因为映射关系在这一步起作用。D3 为应用不同的变换提供了一个构造，不过映射规则还得由你来定。大数值应该映射为更长的条形，还是颜色更浅的圆形？数据聚类（cluster）在 *x* 轴应该按年龄还是按类别排序？世界地图中的国家应该用什么颜色来填充？诸如此类的设计决定完全是你的事。你来挖掘概念、编写规则，D3 来执行——你不用管它怎么执行。（没错，跟 Excel 中那个爱出风头的“图表向导”恰好相反。）

2.2 D3不能做什么

下面这些事 D3 都不能做。

- D3 不能生成预定义的或“事先处理好”的视觉图形。是有意不这么做的。D3 主要用于生成那些解释型的，而非探索型的可视化图形。探索型工具可以帮你发现数据中明显的、有价值的模型。Tableau（http://www.tableausoftware.com/）和 ggplot2（http://ggplot2.org/），这都是探索型工具，能帮你根据相同的数据集生成多个视图。这一点很基础，但却不同于生成数据的解释型表现，即通过数据视图表现出你已经发现的结论。解释型视图约束条件更多，限制也更多，同时也更容易做专做精，而且主要用于传达最重要的信息。D3 擅长生成解释型视图，不擅长探索型视图。（要了解与 D3 类似的其他工具，可以参考本章 2.4 节。）
- D3 不打算支持旧版本的浏览器。这样有助于保持 D3 代码库的干净，避免为支持诸如旧版 Internet Explorer 而加入太多的偏门代码。重点在于通过创建出更有吸引力的工具，同时拒绝旧版浏览器，可以鼓励更多用户升级（而不是延缓这一进程，让更多的人继续使用那些浏览器，然后再延缓……如此恶性循环）。D3 希望我们大家向前看。
- D3 的核心功能不处理像谷歌地图或 Cloudmade 等提供的那些位图格式的地图贴片。D3 最擅长处理矢量图形（SVG 图或 GeoJSON 数据），从一开始就没有打算支持地图贴片。（位图由像素构成，很难在放大或缩小时做到不失真。矢量图则

是由点、直线和曲线——实际上是数据方程式——定义的，因此可以随意放大或缩小而不会失真。）不过，事情现在有了转机，但要配合使用 d3.geo.tile 插件（http://t.cn/zTINvoy）。在这个插件问世之前，通过 D3 来实现地图映射时要么完全依赖 SVG 而不用贴片，要么得通过 D3 在地图贴片基层上面创建 SVG 视觉效果（这时要借助其他库，比如 Leaflet 或 Polymaps，具体请参见本章 2.4 节）。关于如何集成位图贴片和矢量图形，一直是 D3 社区热议的话题。到今天，也没有出现很简单很完美的方案。不过我相信，这一块肯定会有一些突破，或许未来某一天，D3 核心会加入一些全新的贴片处理方法。

- D3 不隐藏你的原始数据。D3 代码在客户端执行（也就是在用户浏览器，而不是 Web 服务器中执行），因此你想要可视化的数据必须发送到客户端。假如你的数据不能共享，就不要使用 D3 了。替代方案是使用专有工具（如 Flash），或将可视化结果预先渲染为静态图片，然后再发送到浏览器。（如果你不想共享数据，那为什么还要把它们可视化呢？可视化的目的就是为了更好地表现数据，与其可视化了之后担心得睡不着觉，还不如一开始就公开化和透明化。）

2.3　起源与背景

第一个浏览器只能渲染静态页面，所谓交互性仅限于单击链接。1996 年，Netscape 在浏览器中内置了 JavaScript 解释器，从而让浏览器在加载页面时，能够解释执行这门脚本语言编写的代码。

这个举措并没有它后来引发的巨变那么惊心动魄，但却让浏览器从被动的显示，进入了交互在线处理动态画面的新时代。这一历史性转变成就了我们今天的页面内交互的 Web。如果没有 JavaScript，就不会有 D3，而基于 Web 的数据可视化也只能局限于提前生成好的、不具备响应能力的 GIF 图。（噢……谢谢，Netscape！）

历史的车轮前进到了 2005 年，这一年 Jeffrey Heer、Stuart Card 和 James Landay 推出 prefuse（http://prefuse.org/），一个通过 Web 呈现的数据可视化工具包。prefuse（字母全部小写）是用 Java 写的，那是一种编译型语言，而且可视化程序要在浏览器中通过 Java 插件运行。（注意，Java 和 JavaScript 是完全不一样的语言，尽管名字上类似。）prefuse 是当时一个突破性的应用，它首次让没有多少经验的编程人员，能够实现基于 Web 的可视化展示。有了 prefuse 之后，Web 上的数据可视化就成了小事一桩。

两年后，Jeff Heer 又推出了 Flare（http://flare.prefuse.org/）。这是一个类似的工具包，编程语言是 ActionScript，就是说可以通过浏览器中的 Flash Player 来查看可视化结果。与 prefuse 类似，Flare 也依赖浏览器插件。Flare 虽然是一个巨大的进步，但随着浏览器的发展，可视化显然不通过插件（而只利用浏览器原生特性）也能实现了。

2009 年，Jeff Heer 搬到斯坦福。在那里，他说服一位刚毕业的学生 Mike Bostock，共同在斯坦福的 Vis Group（http://vis.stanford.edu/）开发了 Protovis（http://mbostock.github.io/d3/tutorial/protovis.html），那是一个基于 JavaScript 的可视化工具包，只依赖原生的浏览器技术。（如果你用过 Protovis，一定要参考 Mike 的这篇“For Protovis Users”，网址：http://mbostock.github.com/d3/tutorial/protovis.html。）

Protovis 简化了生成可视化图形的工作，即使是没有编程经验的人都可以上手。但它要借助一个抽象的表现层，尽管设计师可以使用 Protovis 语法来控制这一层，可调试很不方便，因为使用的不是标准方法。

2011 年，Mike Bostock、Vadim Ogievetsky 和 Jeff Heer 正式推出 D3（http://vis.stanford.edu/papers/d3），作为下一代 Web 可视化工具。与 Protovis 不同的是，D3 直接操作网页文档。因此，调试就方便了，尝试不同的方案也更容易，而且展示视觉效果的可能性也更多了。唯一的缺点是学习门槛有点高，不过本书会尽可能解决这个问题。此外，你通过学习 D3 掌握的所有技术，即使在数据可视化这个领域之外，也将是非常有用的。

无论你熟悉上面提到的任何一个突破性的工具，一定都会认可 D3 纯正的血统。如果你对 D3 底层的设计思想感兴趣，强烈建议你看一看 Mike、Vadim 和 Jeff 在 InfoVis 上发表的论文“D^3 : Data-Driven Documents”（http://vis.stanford.edu/files/2011-D3-InfoVis.pdf），其中清晰地分析了这种工具的必要性。这篇论文浓缩了他们在学习和开发可视化工具几年间的心血。

2.4　替代方案

D3 也不是适合所有项目。有时候，可能你只想马上生成一张图表，没有时间自己编写代码。或者，你想支持旧版本浏览器，因此不能依赖于 SVG 等较新的技术。

在这种情况下，最好是知道还有其他什么选择。以下我就来简单介绍一下 D3 的部分替代方案，也许不全，但它们的共同特点是都采用了 Web 标准技术（主要是 JavaScript），而且可以免费下载使用。

2.4.1　简易图表

- DataWrapper
  一个非常漂亮的在线服务，上传数据并快速生成图表后，就可以到处使用或将其嵌入在自己的站点中。这个服务最初定位于专栏记者，而实际上任何人都可以使用。DataWrapper 在新版本浏览器中可以显示动态图表，而在旧版本浏览器中则

显示静态图片。（太聪明了！）你也可以下载代码在自己的服务器上运行。地址：http://datawrapper.de/。

- Flot
  一个基于 jQuery 的绘图库，使用 HTML 的 canvas 元素，也支持旧版本浏览器（甚至 IE6）。它支持有限的视觉形式（折线、散点、条形、面积），但使用很简单。地址：http://www.flotcharts.org/。

- Google Chart Tools
  由早期的 Image Charts API 发展而来的 Google Chart Tools，可以用来生成不少标准的图表，也支持旧版本的 IE。地址：https://developers.google.com/chart/。

- gRaphaël
  基于 Raphaël（参见本节后面）的一个图表库，支持旧版本浏览器（包括 IE6）。与 Flot 相比，它更灵活，而且据说还要更漂亮一些。地址：http://g.raphaeljs.com/。

- Highcharts JS
  JavaScript 图表库，包含一些预定义的主题和图表。它在最新浏览器中使用 SVG，而在旧版本 IE（包括 IE6 及更新版本）中使用后备的 VML。这个工具只对非商业用途免费。地址：http://www.highcharts.com/。

- JavaScript InfoVis Toolkit
  简称 JIT，它提供了一些预设的样式可用于展示不同的数据，包括很多例子，而文档的技术味道太浓。如果你喜欢它的预设样式，可以选择它，但浏览器支持情况不太清楚。地址：http://philogb.github.com/jit/。

- jqPlot
  jQuery 绘图插件，只支持一些简单的图表，适合不需要自定义样式的情况。jqPlot 支持 IE7 及更新版本。地址：http://www.jqplot.com/。

- jQuery Sparklines
  可生成波形图的 jQuery 插件，主要是那些可以嵌在字里行间的小条形图、折线图、面积图。支持大多数浏览器，包括 IE6。地址：http://omnipotent.net/jquery.sparkline/#s-about。

- Peity
  jQuery 插件，可生成非常小的条形图、折线图和饼图，只支持较新版本的浏览器。再强调一遍，它能生成非常小又非常精致的小型可视化图表，可爱程度加 10 分。地址：http://benpickles.github.com/peity/。

- Timeline.js

  专门用于生成交互式时间线的一个库。不用编写代码，只用其代码生成器即可。定制的空间不大，但时间线可不是那么容易做的。Timeline.js 只支持 IE8 及之后的版本。地址：http://timeline.verite.co/。

- YUI Charts

  雅虎 YUI（Yahoo! User Interface Library）的 Charts 模块，可用于创建简单的图表，支持很多浏览器。地址：http://yuilibrary.com/yui/docs/charts/。

2.4.2 图谱可视化

所谓“图谱”，就是具有网络结构的数据（比如 B 连接到 A，A 连接到 C）。

- Arbor.js

  基于 jQuery 的图谱可视化库。就算没用过它，也该看一看它的文档，连它的文档都是用这个工具生成的（可见它有多纯粹、多 meta）。这个库使用了 HTML 的 `canvas` 元素，因此只支持 IE9 和其他较新的浏览器，当然也有一些针对旧版浏览器的后备措施。地址：http://arborjs.org/。

- Sigma.js

  一个非常轻量级的图谱可视化库。无论如何，你得看看它的网站，在页面上方的大图上晃几下鼠标，然后再看看它的演示。Sigma.js 很漂亮，速度也快，同样使用 `canvas`。地址：http://sigmajs.org/。

2.4.3 地图映射

我们要区分一下地图（全部内容都是地图）和地图映射（包括地理位置数据或地理数据，比如传统的地图）。D3 本身也有很多地图映射功能，但下面这些工具最好你也了解一下。

- Kartograph

  Gregor Aisch 开发的一个基于 JavaScript 和 Python 的非常炫的、完全使用矢量的库，它的演示是必看的。最好现在就去看一看。保证你从来没见过这么漂亮的在线地图。Kartograph 支持 IE7 及更新版本。地址：http://kartograph.org/。

- Leaflet

  贴片地图的库，可以在桌面和移动设备上流畅地交互。它支持在地图贴片上显示一些 SVG 数据层。（参见 Mike 的演示“Using D3 with Leaflet”：http://bost.ocks.org/mike/leaflet/。）Leaflet 支持 IE6（勉强）或 IE7（好得多），当然还有其

他更新版本的浏览器。地址：http://leafletjs.com/。

- Modest Maps
  作为贴片地图库中的老爷爷，Modest Maps 已经被 Polymaps 取代了，但很多人还是喜欢它，因为它体积小巧，又支持 IE 和其他浏览器的老版本。Modest Maps 有很多版本，包括 ActionScript、Processing、Python、PHP、Cinder、openFrameworks……总之，它属于老当益壮那种。地址：http://modestmaps.com/。

- Polymaps
  显示贴片地图的库，在贴片上可以叠加数据层。Polymaps 依赖于 SVG，因此在较新的浏览器中表现很好。地址：http://polymaps.org/。

2.4.4 较原始的方案

以下工具跟 D3 有些类似，都提供了绘制图形的方法，但没有预定义的模板。如果你愿意从头开始，希望得到更大的自由度，可能会对它们感兴趣。

- Processing.js
  Processing 的原生 JavaScript 实现，是新接触编程的艺术家和设计师的梦幻式编程语言。Processing 是 Java 写的，因此 Processing 草图要在网页中显示通常要靠 Java 小程序。有了 Processing.js，常规的 Processing 代码就可以在浏览器中直接运行了。由于使用 `canvas`，所以只适合现代的浏览器。地址：http://processingjs.org/。

- Paper.js
  在 `canavs` 上渲染矢量图形的框架。同样，它的网站也堪称互联网上最漂亮的网站之一，它们的演示做得让人难以置信。（现在就去欣赏一下吧。）地址：http://paperjs.org/。

- Raphaël
  也是一个绘制矢量图形的库，受欢迎的原因是语法具有亲和力，而且支持老版本浏览器。地址：http://raphaeljs.com/。

2.4.5 三维图形

说来也怪，D3 不擅长 3D，因为浏览器从一开始就是二维的东西。但随着它对 WebGL 的支持越来越完善，在网页中显示 3D 图形也会渐渐成为一种趋势。

- PhiloGL
  专注于 3D 可视化的一个 WebGL 框架。地址：http://www.senchalabs.org/philogl/。

- Three.js

  能帮你生成任何 3D 场景的一个库，谷歌 Data Arts 团队出品。它的演示可以让人整整一天都沉浸其中，兴奋不已。地址：http://mrdoob.github.com/three.js/。

2.4.6 基于D3的工具

如果你使用 D3，但又不想写代码，可以考虑下面这些基于 D3 的工具。

- Crossfilter

  一个可以操作大型、多元数据集的库，主要作者是 Mike Bostock。非常适合把你的“大数据”塞到相对小的浏览器里，地址：http://square.github.com/crossfilter/。

- Cubism

  时间序列数据可视化的 D3 插件，也是 Mike Bostock 写的。（我非常喜欢其中的演示。）地址：http://square.github.com/cubism/。

- Dashku

  用于实时更新在线控制板和小部件的在线工具，作者是 Paul Jensen。地址：https://dashku.com/。

- dc.js

  这里的“dc”是 dimensional charting（维度图表）的简写，因为这个库是专门为探索大型、多维数据集而进行优化的。地址：http://nickqizhu.github.com/dc.js/。

- NVD3

  可重用的 D3 图表。NVD3 提供了很多漂亮的示例，不用像在 D3 里那样编写代码就可以定制很多效果。地址：http://nvd3.org/。

- Polychart.js

  更多可重用的图表，可选择的图表类型非常之多。Polychart.js 只对非商业用途免费。地址：http://polychart.com/。

- Rickshaw

  显示时间序列数据的一个工具包，提供了很多定制选项。地址：http://code.shutterstock.com/rickshaw/。

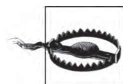
- Tributary

  实时测试 D3 代码的一个好工具，作者是 Ian Johnson。地址：http://tributary.io/。

第3章

技术基础

本章将介绍一些相关的基本概念，熟悉这些概念对掌握 D3 大有裨益，当然也能减少挫折感。嗯，就当本章是一个 Web 开发的入门教程吧。



请大家留意一下，本章的知识密度很高，浓缩了多年来形成的 Web 开发知识，而且都不是专门针对 D3 的。建议大家只捡那些自己不知道的内容看，其他的可以跳过。到后面再遇到什么问题时，可以再回来。

3.1 Web（万维网）

如果你从来没做过网页，那现在必须得想一想平常人大都不屑一顾的一个问题了：Web 的原理是什么？

我们一般把 Web 看成一堆内部相互链接的页面，而实际上它是服务器与客户端（浏览器）之间你来我往的对话。

如果你单击了一个链接或者在浏览器地址栏输入了一个网站，那么就会出现下面这个场景（这个场景每天都要上演无数次）：

> 客户端：我想知道somewebsite.com上有什么，去登门造访一下吧，看看有什么新消息。[互联网连接静静地建立起来]
>
> 服务器：你好，新来的Web客户端！我是托管somewebsite.com的服务器。你想看哪个页面？

客户端：现在还早，我想看一看somewebsite.com/news/下面的页面中有什么新闻。

服务器：没问题，稍等。

（一串数字代码从服务器传输到浏览器。）

客户端：收到了。谢谢！

服务器：不客气！我非常愿意跟您在线多聊一会儿，不巧又来新请求了。再见！

客户端连接到服务器并发送请求，而服务器响应数据。可到底什么是服务器，什么是客户端？

服务器是连在互联网上的计算机，运行着 Web 服务器软件。之所以称其为 Web 服务器，正是因为它们会根据请求来提供网页，像个服务员似的。服务器一般 24 小时运行，而且永远在线。有时候，Web 开发人员也会在本地计算机上运行 Web 服务器软件，当然你也可以。本地是什么意思？就是你这台电脑啊，远程指就是其他电脑，或者说除了你眼前这台电脑之外的其他电脑。

有很多现成的服务器软件，比如 Apache。Web 服务器软件可没那么好看，也没人愿意看。

相对而言，Web 浏览器就赏心悦目多了，我们每天都盯着它看个没完。大家一般都听说过 Firefox、Safari、Chrome，还有 Internet Explorer，这些都是浏览器，也就是客户端。

理论上讲，每个网页都有一个唯一的 URL（Uniform Resource Locator，统一资源定位符），也叫 URI（(Uniform Resource Identifier，统一资源标识符）。大多数人不知道 URL 是怎么回事，但看见就能认出来。URL 一般以 www 开头，比如 http://www.calmingmanatee.com。但服务器如果做过配置，那么“www.”这几个字符完全可以省掉不写。

完整的 URL 由以下 4 部分构成：

- 通信协议，如 HTTP 或 HTTPS；
- 资源所在的域名，如 calmingmanatee.com；
- 端口号，表示要连接到服务器的哪个端口上；
- 其他定位信息，如所请求文件的路径或查询参数。

于是，一个比较完整的 URL 就是这个样子的：http://alignedleft.com:80/tutorials/d3/。

平时一般不用指定端口号，因为浏览器默认会连接服务器的 80 端口。因此，前面的 URL 与下面这个功能完全一样：http://alignedleft.com/tutorials/d3/。

URL 开头的协议名跟后面的域名之间是一个冒号和两个斜杠。为什么是两个斜杠？没有为什么，这就么规定的。可以算个失误吧，Web 的发明人也为这个失误后悔不迭。

HTTP 代表 Hypertext Transfer Protocol（超文本传输协议），是服务器与客户端之间传输 Web 内容最常用的协议。HTTPS 后面的“S”代表 Secure（安全）。因此，HTTPS 一般用于传输加密信息，比如在线交易或电子商务。

下面我们简单描述一个人访问某个站点时的过程。

1. 用户打开自己的浏览器，在地址栏中输入 URL，例如 alignedleft.com/tutorials/d3/。因为没有指定协议，所以浏览器会使用默认的 HTTP 协议，在 URL 前面补上“http://”。
2. 浏览器尝试通过互联网连接 alignedleft.com 所在的服务器，连接其 80 端口（这是 HTTP 连接的默认端口）。
3. 与 alignedleft.com 关联的服务器同意连接，并准备接收浏览器的请求。（“我会等你一晚上，不见不散。”）
4. 浏览器请求访问位于目录 /tutorials/d3/ 下的页面。
5. 服务器把那个页面的 HTML 内容发给浏览器。
6. 浏览器收到 HTML 后，根据其中引用的其他文件（包括 CSS 样式表和图片）再聚合并显示出完整的页面。为此它还要再连接到同一台服务器，每次请求并取得一个文件。
7. 服务器响应，根据请求发回每个文件。
8. 网页文档传输完毕。浏览器履行它最费劲的职责——渲染内容。首先通过解析 HTML 确定内容的结构，然后根据 CSS 选择符为匹配的元素应用样式，最后把图片插入到页面中，并执行 JavaScript 代码。

或许你不敢相信，每单击一次链接都会经历上述步骤。这些步骤比大多数人想象的要复杂，但却是理解客户端 / 服务器对话的基础，也是理解 Web 运行原理的基础。

3.2 HTML

HTML（Hypertext Markup Language，超文本标记语言）用于向浏览器说明内容的结构。HTML 保存在以 .html 为扩展名的纯文本文件中。下面就是一个简单的 HTML 文档。

```
<!DOCTYPE html>
<html>
    <head>
         <title>Page Title</title>
    </head>
```

```
    <body>
        <h1>Page Title</h1>
        <p>This is a really interesting paragraph.</p>
    </body>
</html>
```

HTML 本身不算简单，历史也比较悠久。本节只介绍 HTML 的最新版本（正式称呼叫 HTML5），而且只介绍那些与 D3 相关的内容。

3.2.1 内容和结构

HTML 的核心功能就是让你“标记”内容，进而给出结构。比如下面这段文本：

> Amazing Visualization Tool Cures All Ills A new open-source tool designed for visualization of data turns out to have an unexpected, positive side effect: it heals any ailments of the viewer. Leading scientists report that the tool, called D3000, can cure even the following symptoms: fevers chills general malaise It achieves this end with a patented, three-step process. Load in data. Generate a visual representation. Activate magic healing function.

只要读上两行，就能知道这是一个很激动人心的新闻报道。可由于内容没有分出结构来，所以读起来很费劲。而添加了结构之后，就可以区分哪里是标题，哪里是正文。

> **Amazing Visualization Tool Cures All Ills**
>
> A new open-source tool designed for visualization of data turns out to have an unexpected, positive side effect: it heals any ailments of the viewer. Leading scientists report that the tool, called D3000, can cure even the following symptoms:
>
> - fevers
> - chills
> - general malaise
>
> It achieves this end with a patented, three-step process.
>
> 1. Load in data.
> 2. Generate a visual representation.
> 3. Activate magic healing function.

文字还是那些文字，但加上结构之后，可读性就不一样了。

HTML 就是为内容添加语义结构（或者说层次、关系和含义）的这么一种手段。（HTML 不考虑文档的外观，那是 CSS 的事。）如果把上面分出结构的内容用语义来描述，就是下面这样。

标题

段落文本

- 无序列表项
- 无序列表项
- 无序列表项

段落文本

1. 有序列表项
2. 有序列表项
3. 有序列表项

这就是我们通过 HTML 标记给内容添加的结构。

3.2.2 通过元素来添加结构

所谓“标记”，就是通过给内容添加标签来创建元素的过程。HTML 标签以 < 开头，以 > 结束，比如表示段落文本的 `<p>`。标签一般都成对出现，一个开始标签和一个结束标签就在文档中创建了一个元素。

结束标签用一个斜杠表示元素的关闭或结束，比如 `</p>`。如果要标记一个段落元素，那么可以像下面这样写：

```
<p>This is a really interesting paragraph.</p>
```

有些元素是可以嵌套的。比如，可以使用 em 元素为文本增加强调的语义。

```
<p>This is a <em>really</em> interesting paragraph.</p>
```

嵌套元素在文档中会形成层次。对上面的例子而言，em 就是 p 的子元素，因为它被 p 包含着。（相应地，p 是 em 的父元素。）

在元素相互嵌套的时候，子元素不能超出父元素之外，因为这样就会破坏层次，比如：

```
<p>This could cause <em>unexpected</p>
<p>results</em>, and is best avoided.</p>
```

有些标签永远不会成对出现，比如指向一张图片的 img。虽然 HTML5 不再强制要求，但你经常也会看到这种标签的自关闭写法，也就是在末尾的 > 之前加一个斜杠：

```
<img src="photo.jpg" />
```

HTML5 之后，这种自关闭的标签就变成了可选的，因此下面的代码跟前面的代码

功能一样：

```
<img src="photo.jpg">
```

3.2.3　常用元素

HTML 有上百个元素，我们只介绍最常用的。后续章节还会介绍其他一些元素。（要了解所有 HTML 元素，请参考全面的 Mozilla Developer Network，网址：https://developer.mozilla.org/en/HTML/Element。）

- `<!DOCTYPE html>`
  这是标准的文档类型声明，必须在文档的第一行。

- `html`
  包含文档中的所有 HTML 内容。

- `head`
  文档的头部，包含所有文档的元数据。比如标题和对外部样式表、脚本的引用。

- `title`
  文档的标题。浏览器会把这个元素的内容显示在窗口标题栏中，并在收藏网页时使用这个标题。

- `body`
  所有不包含在 `head` 中的内容，都包含在 `body` 中。这里面的内容是可以在网页中看到的。

- `h1`、`h2`、`h3`、`h4`
  用于标记不同级别的标题。`h1` 代表顶级标题，`h2` 代表二级标题，依此类推。

- `p`
  段落！

- `ul`、`ol`、`li`
  `ul` 用于标记无序列表，也就是带项目符号的列表。`ol` 用于标记有序列表，也就是带编号的列表。`ul` 和 `ol` 都包含 `li` 元素，用于标记列表项。

- `em`
  表示强调，一般显示为斜体。

- `strong`
  表示额外强调，一般显示为粗体。

- `a`

  链接。一般显示为带下划线的蓝色文本，可以另行设置。

- `span`

  任意文本，一般都包含在 p 这样的大容器元素中。

- `div`

  任意文本块。用于分组相关元素。

我们前面的例子就是使用这些元素来区分语义结构的：

```
<h1>Amazing Visualization Tool Cures All Ills</h1>
<p>A new open-source tool designed for visualization of data turns out
to have an unexpected, positive side effect: it heals any ailments of the
viewer. Leading scientists report that the tool, called D3000, can cure
even the following symptoms:</p>
<ul>
    <li>fevers</li>
    <li>chills</li>
    <li>general malaise</li>
</ul>
<p>It achieves this end with a patented, three-step process.</p>
<ol>
    <li>Load in data.</li>
    <li>Generate a visual representation.</li>
    <li>Activate magic healing function.</li>
</ol>
```

在浏览器中打开这个页面，可以看到如图 3-1 所示的结果。

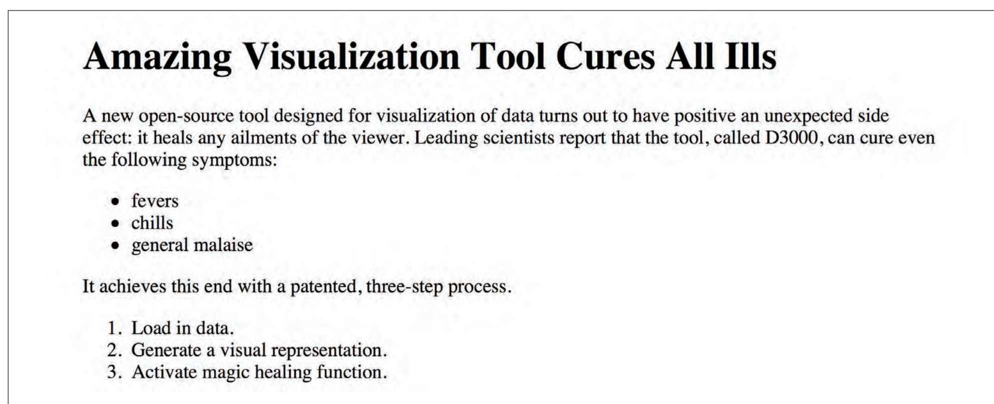


图 3-1：简单的 HTML 在浏览器中默认的效果

我们到现在只介绍了给文档标记语义结构，还没有涉及视觉样式，比如颜色、字体大小、缩进或行间距。如果没有这种指令，浏览器就会使用默认的样式。当然，默

认样式中规中矩，不会好看到哪里去。

3.2.4 属性

可以为任何 HTML 元素指定一些属性，形式如下（在开始标签中）：

```
< 标签名 属性 =" 值 "></ 标签名 >
```

属性名后面是等于号，然后是放在双引号中的属性值。

不同的元素拥有不同的属性。比如，`a` 是一个链接元素，有 `href` 属性，这个属性的值就是链接的 URL。（href 是 HTTP reference 的简写。）

```
<a href="http://d3js.org/">The D3 website</a>
```

有些属性可以指定给任何元素，比如 `class` 和 `id`。

3.2.5 类和ID

`class` 和 `id` 是最有用的两个属性，因为通过它们可以找到特定的内容。而且，CSS 和 JavaScript 代码也依赖它们定位元素。比如：

```
<p>Brilliant paragraph</p>
<p>Insightful paragraph</p>
<p class="awesome">Awe-inspiring paragraph</p>
```

这里有三个段落，但只有一个真“不得了‘（awesome），因为我给它添加了 `class="awesome"` 属性。这样，第三段就归到了“不得了”（awesome）的元素一类。通过这个类，就可以选择并操作它。（稍后再介绍具体做法。）

可以给一个元素指定多个类，多个类名间以空格分隔；也可以给多个元素指定一个类：

```
<p class="uplifting">Brilliant paragraph</p>
<p class="uplifting">Insightful paragraph</p>
<p class="uplifting awesome">Awe-inspiring paragraph</p>
```

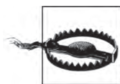
这样，所有三个段落就都“令人振奋”（`uplifting`）了，但只有最后一个段落既“令人振奋”又“不得了”。

ID 的使用方法类似，但每个元素只能有一个 ID，而且这个 ID 在整个页面中也只能出现一次。例如：

```
<div id="content">
    <div id="visualization"></div>
    <div id="button"></div>
</div>
```

在某个元素比较特殊的情况下，使用 ID 比较合适。比如，想让一个 div 像按钮一样，或者作为页面中其他内容的容器。

一般来说，如果页面中只有这么一个元素，就可以给它指定 id 属性。否则，就使用 class 属性。



类和 ID 的值不能以数字开头，而必须以字母开头。比如，id="1" 不合规则，而 id="item1" 就可以。写错了，浏览器也不报错，但你的代码就运行不了了。为了找到问题所在，恐怕都得疯掉。

3.2.6 注释

随着代码越来越多，越来越复杂，最好的做法是添加注释。注释就是你描述代码用途和实现方式的一些备注。如果你跟我一样，几个星期之后才会重新看一看代码，那很可能早就忘了以前是怎么想的了。注释就能起到提醒的作用，在未来给你提供一个向导。

HTML 中的注释是这样来写的：

```
<!-- 这里是注释的内容 -->
```

位于 <!-- 和 --> 之间的内容都会被浏览器忽略掉。

3.3 DOM

DOM（Document Object Model，文档对象模型）指的是 HTML 标签的层次结构。每一对 HTML 标签（有时候则是一个标签）都是一个元素，对这些元素，我们一般用拟人化的方法来称呼它们，比如：父元素、子元素、同胞元素、祖先元素和后代元素。举个例子，对下面的 HTML 来说：

```
<html>
    <body>
         <h1>Breaking News</h1>
        <p></p>
    </body>
</html>
```

body 是其子元素 h1 和 p 的父元素，而 h1 和 p 则互为同胞元素。页面中的所有元素都是 html 的后代元素。

浏览器通过解析 DOM 来操作页面内容。编码的人在实现可视化时，最关心的是 DOM。因为我们的代码必须在 DOM 层次中寻找元素，然后为它们应用样式和行

为。如果不想让所有 `div` 元素都显示成蓝色，那就必须想办法选择 `class` 为 `sky` 的 `div`，把它们设置成蓝色。

3.4　开发者工具

最早的时候，Web 开发的工作流程是这样的：

1. 在文本编辑器中写代码；
2. 保存文件；
3. 切换到浏览器；
4. 刷新页面，看代码是否有效。
5. 如果代码无效，猜一猜是哪出了问题，然后返回第 1 步。

浏览器是出了名的“密不透风”的黑盒子，其渲染引擎怎么工作很难预测，这就给代码调试带来了无尽的麻烦。（说真的，1999 年年底 2000 年年初的时候，我就因此烦透了。）好在，我们生活在一个文明的时代。现代的浏览器都内置了开发者工具，能够把浏览器内部的工作过程展示出来，让我们看到后台的秘密。

说这些的意思就是告诉大家，开发者工具十分重要。你在写代码的时候，经常需要用它来测试代码，而在遇到问题时，还要借助它找到原因所在。

下面我们就介绍开发者工具的一个最简单的用途：查看 HTML 页面的源代码（参见图 3-2）。

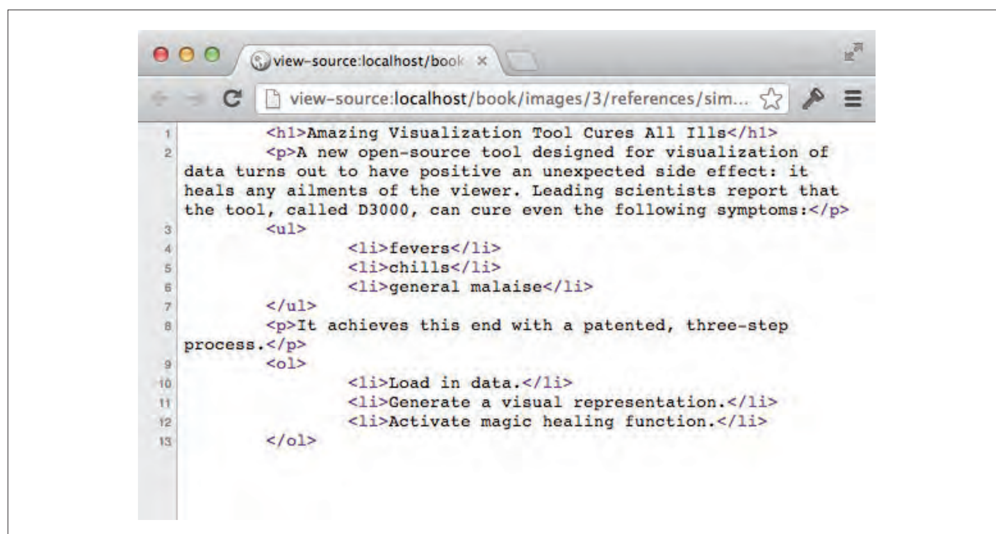


图 3-2：在新窗口中查看源代码

所有浏览器都支持这个功能，只不过它们可能会把这个选项放到不同的地方。在 Chrome 23.0 中，是“工具 > 查看源代码”。在 Firefox 17.0 中，是“工具 > Web 开发者 > 页面源代码”。在 Safari 6.0 中，是“开发 > 显示源代码”（前提是必须在“Safari > 偏好设置 > 高级”中，把“开发”菜单设置为可见）。接下来，无论你使用什么浏览器，我都假设你使用的是它的最新版本。

如图 3-2 所示，其中显示了 HTML 源代码。如果有 D3 或 JavaScript 代码执行，那当前 DOM 就会大不相同。

好在浏览器的开发者工具可以帮我们查看 DOM 的当前状态。同样，各个浏览器都提供了自己的开发者工具。在 Chrome 中，可以在“工具 > 开发者工具”中找到。在 Firefox 中，可以试试“工具 > Web 开发者”。在 Safari 中，首先要启用开发者工具（在“偏好设置 > 高级”下），然后，在“开发”菜单中选择“显示 Web 检查器”。在任何浏览器中，都可以使用相应的键盘快捷键（就在前面提到的菜单项旁边可以找到），或者通过右键单击并选择“检查元素”也能调出开发者工具。

直到最近，Safari 和 Chrome 使用的都是相同的开发者工具。从 Safari 6.0 开始，苹果完全重新设计了自己的开发工具，但让粉丝们很失望。（新工具非常难用，而且有这种感觉的不只我一人。）无论你使用什么浏览器，无论你的界面看起来跟我这个屏幕截图有多不一样，其实它们的功能都非常相近。

图 3-3 显示了 Chrome 的 Web 检查器的 Elements（元素）选项卡。在这里，可以看到 DOM 的当前状态。因为我们的代码会动态修改 DOM 元素，所以这个选项卡非常有用。通过这个窗口，可以随时监控元素的变化。

仔细看一看，你就会发现 HTML 源代码和 DOM 层次的区别。而且，Chrome 还帮我们生成了必要的 `html`、`head` 和 `body` 元素。（我总是很懒，不愿意在 HTML 中写这些标签。）

还有一点，为什么我只讲 Chrome、Firefox 和 Safari 呢？为什么不提 IE、Opera 和其他浏览器呢？首先，最好使用对 Web 标准支持得最完善的浏览器。IE 虽然从第 9 个版本到第 10 版本有了很大进步，但 Chrome、Firefox 和 Safari 对标准的支持仍然是最好的，而且升级很频繁。

其次，我们得经常使用开发者工具，必须选择那些工具好用的浏览器。我个人很喜欢 Safari 6.0 之前的开发者工具，之后的就不好用了。所以我就只能选择 Chrome 和 Firefox 的开发者工具。建议大家都试一试，选择一个最适合自己的。

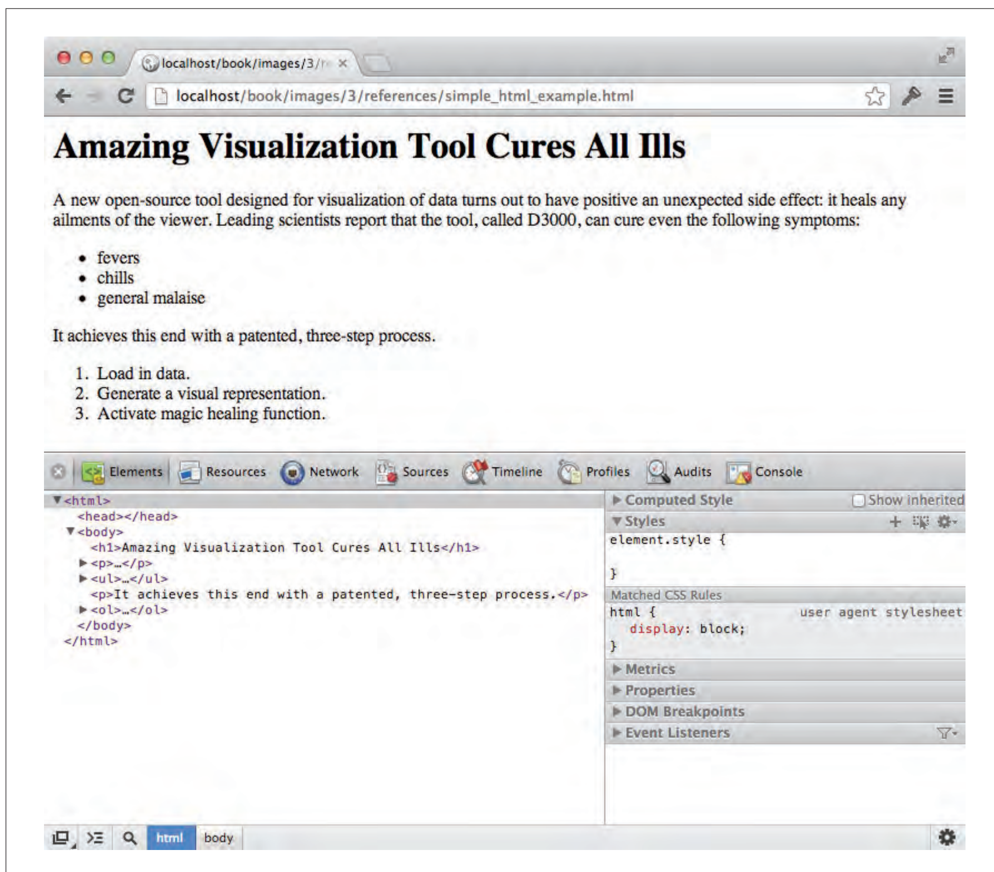


图 3-3：Chrome 的元素检查窗口

3.5　渲染与盒模型

渲染，就是浏览器在解析 HTML 并生成 DOM 后，对 DOM 应用视觉规则并将像素绘制到屏幕的过程。

要知道浏览器如何渲染内容，最重要的是记住一点：浏览器把一切都看成盒子（box）。`p`、`div`、`span`，都是盒子，因为它们在屏幕这个二维空间里都是矩形，具有任何矩形的特征，如宽度、高度、位置。就算有些元素看起来有圆角或者形状不规则，放心，在浏览器眼里，它们也都是一个个的矩形盒子。

不相信？在 Web 检查器的帮助下，你可以看到这个盒子。只要把鼠标移动到任意元素上，与之对应的盒子就会高亮显示出来，如图 3-4 所示。

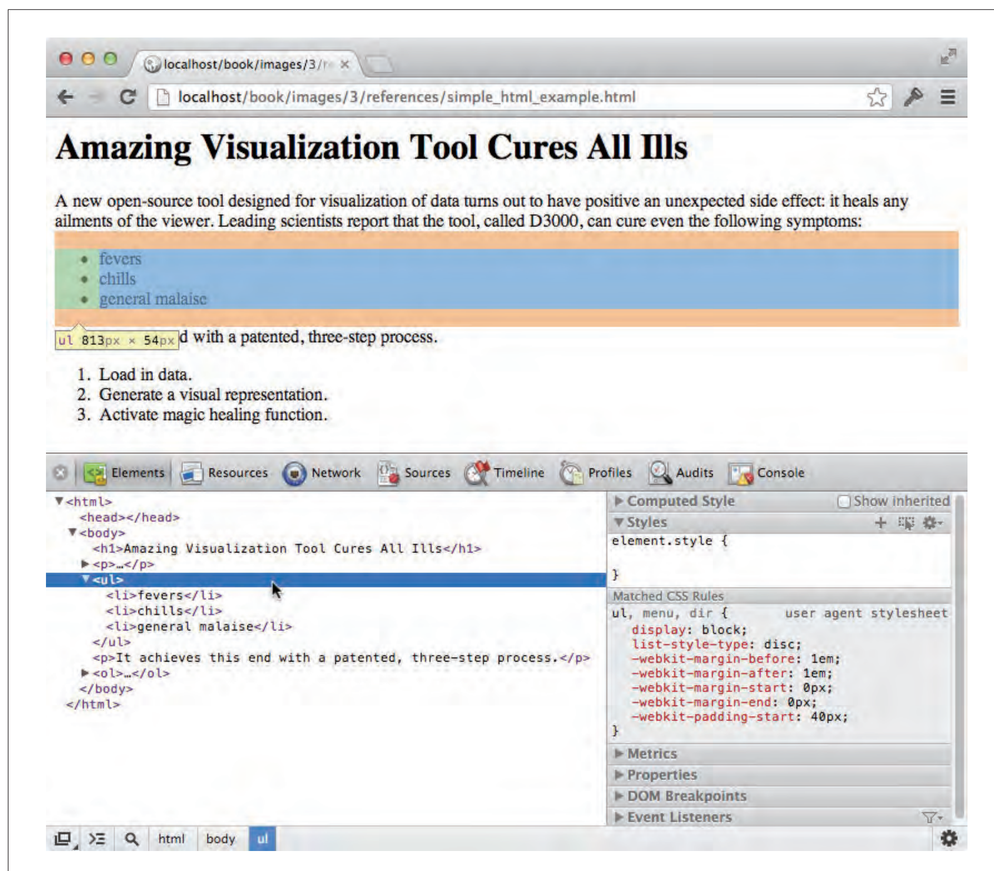


图 3-4：高亮元素盒子的检查器

图中的无序列表 `ul` 有很多方面的信息。首先，这个列表总的大小（宽度和高度）显示在了左下角黄色的小提示框中。其次，这个列表在 DOM 中的位置，通过检查器左下角也能看出来：html > body > ul。

另外，`ul` 盒子扩展到了与整个窗口同宽，因为它是一个块级元素。（可以看看检查器右侧“Computed：Style”中，有一条 `display:: block`。）与之相对的是行内元素，顾名思义，就是构成文本行的元素，而不是像块级元素那样上下堆叠。常见的行内元素包括 `strong`、`em`、`a` 和 `span`。

默认情况下，块级元素会扩展到填满父元素，从而把后面的同胞元素挤到自己脚下。行内元素不会扩展，它们相互并肩排列，一行之中鳞次栉比。（思考一下：你想成为什么元素？）

3.6 CSS

CSS（Cascading Style Sheets，层叠样式表）控制 DOM 元素的视觉外观。下面就是一条简单的 CSS 样式：

```
body {
    background-color: white;
    color: black;
}
```

CSS 样式由选择符和属性组成。选择符后面跟着属性，但被一对花括号所包围。属性和值由冒号分隔，每个属性声明以分号结尾。

```
选择符 {
    属性 : 值 ;
    属性 : 值 ;
    属性 : 值 ;
}
```

相同的属性可以应用给多个选择符，只要用逗号分隔选择符即可，比如：

```
选择符 A,
选择符 B,
选择符 C {
    属性 : 值 ;
    属性 : 值 ;
    属性 : 值 ;
}
```

举个例子，假设你想让段落 `p` 和列表项 `li` 拥有相同的文字大小、行高和颜色，可以这样写：

```
p,
li {
    font-size: 12px;
    line-height: 14px;
    color: orange;
}
```

所有这些放到一块（选择符和带花括号的属性）就叫做一条 CSS 规则。

3.6.1 选择符

D3 使用 CSS 式的选择符来标识要操作的元素，因此理解怎么使用选择符至关重要。

CSS 中的选择符用于标识要应用样式的元素，有很多种，我们这里只介绍其中最简单的几种。

1. 类型选择符

这种选择符最简单，就是匹配同名 DOM 元素的标签名：

```
h1          /* 选择所有一级标题       */
p           /* 选择所有段落          */
strong      /* 选择所有 strong 元素  */
em          /* 选择所有 em 元素      */
div         /* 选择所有 div 元素     */
```

2. 后代选择符

后代选择符匹配包含在（或“出身于”）另一个元素中的元素。在应用样式的时候，后代选择符的使用率非常高：

```
h1 em       /* 选择包含在 h1 中的 em 元素 */
div p       /* 选择包含在 div 中的 p 元素 */
```

3. 类选择符

类选择符匹配具有指定类的所有元素。类名之前要加一个英文句点，看下面的例子：

```
.caption    /* 选择带 "caption" 类的元素  */
.label      /* 选择带 "label" 类的元素    */
.axis       /* 选择带 "axis" 类的元素     */
```

有些元素可能包含多个类，为此可以把多个类名串起来选择它们，比如：

```
.bar.highlight  /* 选择高亮（hightlight）的条形（bar）*/
.axis.x         /* 选择 x 轴（axix）*/
.axis.y         /* 选择 y 轴（axix）*/
```

`.axis` 可以为两个轴应用样式，而 `.axis.x` 则只能为 *x* 轴应用样式。

4. ID选择符

ID 选择符匹配具有给定 ID 的一个元素。（别忘了，一个 ID 只能在 DOM 中出现一次。）ID 前面要带一个井字号。

```
#header     /* 选择 ID 为 "header" 的元素  */
#nav        /* 选择 ID 为 "nav" 的元素     */
#export     /* 选择 ID 为 "export" 的元素  */
```

选择符可以通过各种组合来达到选择特定元素的目的。比如，可以把两个选择符串起来，选择一个更具体的元素：

```
div.sidebar /* 只选择带有 "sidebar" 类的 div，而不选择带其他类的 div  */
#button.on  /* 只选择带有 "on" 类，且 ID 为 "button" 的元素 */
```

提醒大家一下，DOM 是动态变化的，类和 ID 随时可能被加入，或者被删除，所以你的 CSS 规则只能适用于某些特定的情况。

要了解更多有关选择符的信息，请参考 Mozilla Developer Network（http://mzl.la/V27Mcr）。

3.6.2 属性和值

多个属性和值累积起来，就会构成特定的样式：

```
margin: 10px;
padding: 25px;
background-color: yellow;
color: pink;
font-family: Helvetica, Arial, sans-serif;
```

虽然大家自己能看明白，但还是多说几句吧。每个属性都应匹配不同的信息，比如 `color` 要有一种颜色值，而 `margin` 需要一个长度值（这里单位是 `px`，即像素）。

顺便说一下，颜色可以使用以下几种格式。

- 颜色名：`orange`。
- 十六进制值：`#3388aa` 或 `#38a`。
- RGB 值：`rgb(10, 150, 20)`。
- 带透明通道的 RGB 值：`rgba(10, 150, 20, 0.5)`。

至于所有的属性，在网上很容易找一个表格，因此这里就不再多说了。将来碰上的时候，我们再逐个介绍。

3.6.3 注释

```
/* 噢，差点给忘了，这是 CSS 中注释的格式。
   以斜杠和星号开头，以另一个星号和斜杠
   结尾。中间的所有内容都会被浏览器忽略。
*/
```

3.6.4 引用样式

把 CSS 规则应用给 HTML 文档的方式有三种。

1. 在HTML中嵌入CSS

把 CSS 规则嵌入到 HTML 文档，可以把所有东西都放在一个文件中。在文档的头部，可以把 CSS 代码放在 `style` 元素中。

```
<html>
    <head>
        <style type="text/css">

            p {
```

```
                font-size: 24px;
                font-weight: bold;
                background-color: red;
                color: white;
            }

        </style>
    </head>
    <body>
        <p>If I were to ask you, as a mere paragraph, would you
          say that I have style?</p>
    </body>
</html>
```

这个 HTML 页面和 CSS 规则渲染后的结果如图 3-5 所示。



图 3-5：渲染嵌入 CSS 规则后的效果

嵌入 CSS 规则是最简单的办法。但是，我建议大家把不同的代码（HTML、CSS 和 JavaScript 代码）分别放在不同的文件中。

2. 在HTML中引用外部样式表

可以把 CSS 保存一个单独的纯文本文件中，扩展名为 .css，比如 style.css。然后，在 HTML 文档中通过头部的 `link` 元素引用该外部样式文件即可。

```
<html>
    <head>
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <p>If I were to ask you, as a mere paragraph, would you
          say that I have style?</p>
    </body>
</html>
```

这样渲染后的结果与前面的例子完全一样。

3. 插入行内样式

第三种应用 CSS 样式的方法就是把 CSS 规则直接插入 HTML 元素标签内。为此，要使用 `style` 属性为元素指定规则。CSS 规则要像其他属性值一样，放到一对双引号中。比如下面的代码会得到图 3-6 所示的结果。

```
<p style="color: blue; font-size: 48px; font-style: italic;">
    Inline styles are kind of a hassle</p>
```

Inline styles are kind of a hassle

图 3-6：渲染行内 CSS 规则的效果

因为行内样式是直接应用到具体元素上的，所以就不需要选择符了。

太多的行内样式会导致 HTML 代码混乱，难以理解。可是在需要给某个元素应用特殊效果，而又不方便把它写进更大的样式表文件时，使用行内样式还是可以接受的。后面我们会介绍在 D3 中怎么以编程的方式为元素应用行内样式（一次应用一种样式，很简单的）。

3.6.5　继承、层叠和特指度

在没有为某个元素指定样式的情况下，这个元素的很多样式都是从它的祖先元素继承来的。比如，下面的文档给 div 指定了一些样式：

```
<html>
    <head>
        <title></title>
        <style type="text/css">

            div {
                background-color: red;
                font-size: 24px;
                font-weight: bold;
                color: white;
            }

        </style>
    </head>
    <body>
        <p>I am a sibling to the div.</p>
        <div>
            <p>I am a descendant and child of the div.</p>
        </div>
    </body>
</html>
```

这个页面在渲染后，本来是应用给 div 的样式（红色背景、粗体文本等）也被第二个段落元素继承了（如图 3-7 所示），因为 p 是这个 div 的后代。



图 3-7：继承样式

继承是 CSS 中一个非常重要的特性，子元素因此会拥有父元素的特征。（现实当中不是也一样嘛。）

最后，我想回答大家在心里憋了很久的一个问题：为什么称之为层叠样式表？因为选择符会从上到下按照层叠关系匹配。说具体一点，假设多个选择符都给一个元素应用了样式，那么后定义的规则就会覆盖先定义的规则。比如下面的规则会将文档中所有 p 元素中文本的颜色设定为蓝色，但带有 highlight 类的 p 元素中的文本则是黑色，而且带有黄色背景，如图 3-8 所示。第一条规则通过选择符 p 首先应用，而第二条规则通过选择符 p.highlight 覆盖了不够具体的 p 规则。

```
p {
    color: blue;
}

p.highlight {
    color: black;
    background-color: yellow;
}
```

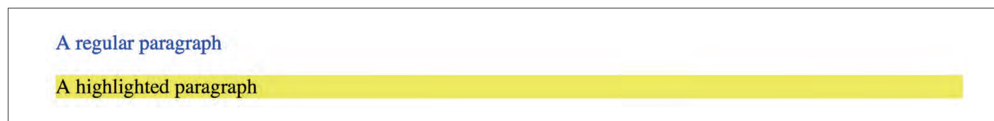


图 3-8：CSS 的层叠与继承

后定义的规则一般会覆盖先定义的规则，但也不全是。关键是要看每个选择符的特指度（specificity）。选择符 p.highlight 即使被放在第一条规则的位置上，它也会覆盖选择符 p，因为它是一个更具体的选择符。假如两个选择符具有相同的特指度，那么这时候后定义的才会胜出。

特指度也是 CSS 中最不好理解的一个地方。计算选择符特指度的规则一两句话讲不明白，所以干脆我们不在这里讲了。为了避免以后麻烦，最好保证自己的选择符清晰、好理解。总的原则是把通用选择符放在最前面定义，而把更具体的选择符放在后面定义，这样就够了。

3.7 JavaScript

JavaScript 是动态脚本语言，通过操作 DOM 动态修改页面。前面提到过，学习 D3 也是学习 JavaScript。更多内容我们会在用到的时候再讲，本节只介绍一些基础知识。

3.7.1 Hello, Console

我们通常都把 JavaScript 代码（或“脚本”）放在一个文本文件中，然后通过网页让浏览器加载这个文件。实际上，也可以直接在浏览器的控制台（Console）中输入 JavaScript 代码。这是一种测试代码的直接而简单的方式，也可以在其中调试代码。总之，控制台是观察代码运行情况的一个基本工具。

在 Chrome 中，按 F12 调出开发者工具，然后切换到“Console”选项卡（或直接按 Ctrl+Shift+J）。在 Firefox 中，按 Shift+F2 打开“开发者工具栏”，然后选择“Web 控制台”。在 Safari 中，按 Ctrl+Alt+C 打开“控制台”，参见图 3-9。

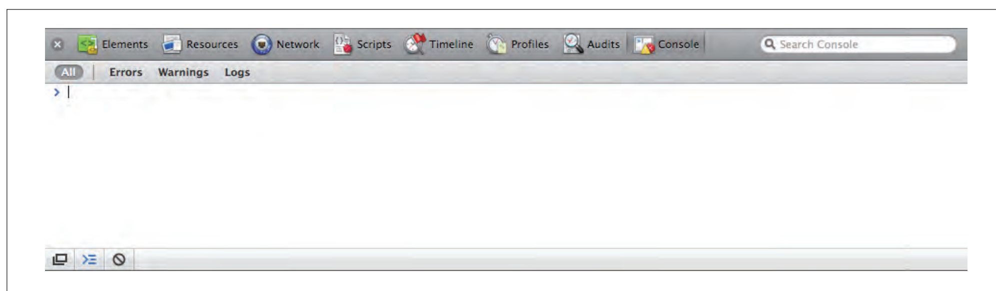


图 3-9：新鲜出炉的 JavaScript 控制台……好好品尝！

在控制台中每次只能输入一行代码，而且会返回你输入的结果。比如，输入数值 7，控制台就会返回 7。一点都不迟疑。

有时候，你需要自己的脚本能把值自动输出到控制台，以便实时观测代码执行过程。为此，可以使用如下代码：`console.log("something");`。

下面就跟着我在控制台中输入示例，看看结果吧。

3.7.2 变量

变量是数据的容器。一个简单的变量可以保存一个值：

```
var number = 5;
```

在这条语句中，`var` 表示要声明一个新变量，而变量的名字是 `number`。等于号是

JavaScript 中的赋值操作符，它先取得右边的值（5），然后把这个值赋给左边的变量（number）。因此，在看到下面这样的代码时

```
defaultColor = "hot pink";
```

可以试着不把等于号理解成“等于”，而是“设置成”。于是，上面的赋值语句就可以读作“把变量 defaultColor 设置成桃红色（"hot pink"）”。

刚才我们看到了，变量可以保存数值，也可以保存文本字符串（之所以叫“字符串”，是因为它们是由一个个的字符串起来的）。字符串必须用引号引起来。除此之外，变量还可以保存布尔值 true 或 false：

```
var thisMakesSenseSoFar = true;
```

另外，在 JavaScript 中，语句是以分号结尾的。

好了，现在可以在控制台中试着声明一些变量。比如，输入 var amount = 200，然后按回车，在下一行继续输入 amount，回车。你会看到控制台返回了 200，这说明 JavaScript 记住了你赋给变量 amount 的值！

3.7.3 其他数据类型

变量保存着数据，是所有数据结构的基础，复杂的数据结构都是由简单的变量组合而成的。

下面我们再介绍几种复杂的数据类型，有数组、对象和对象的数组。如果你觉得可行，现在可以跳过下面的内容，等到考虑向 D3 中加载数据时再回来补课。

1. 数组

数组由一系列值组成，这些值方便地保存在一个变量中。相反，如果用不同的变量记录多个值会很不方便：

```
var numberA = 5;
var numberB = 10;
var numberC = 15;
var numberD = 20;
var numberE = 25;
```

如果写成数组，那么表示这些值的形式就简单多了。JavaScript 中的方括号（[]）代表数组，每个值用英文逗号分隔：

```
var numbers = [ 5, 10, 15, 20, 25 ];
```

数组在数据可视化中是无处不在的，所以你会对它非常熟悉的。要取得（访问）数组中的值，也要使用方括号：

```
numbers[2]  // 返回 15
```

方括号中的数字代表数组中值的位置，但这个位置是从零开始计数的。因此，数组中的第一个位置用 0 表示，第二个位置用 1 表示，依此类推：

```
numbers[0]  // 返回 5
numbers[1]  // 返回 10
numbers[2]  // 返回 15
numbers[3]  // 返回 20
numbers[4]  // 返回 25
```

把数组想象成一个由行和列组成的表格可能更好理解：

位　置	值
0	5
1	10
2	15
3	20
4	25

数组可以包含任何类型的数据，不光是整数：

```
var percentages = [ 0.55, 0.32, 0.91 ];
var names = [ "Ernie", "Bert", "Oscar" ];

percentages[1]  // 返回 0.32
names[1]        // 返回 "Bert"
```

尽管不推荐，但不同类型的数据照样可以保存在同一个数组中：

```
var mishmash = [ 1, 2, 3, 4.5, 5.6, "oh boy", "say it isn't", true ];
```

2. 对象

数组很适合保存一系列的值，但对于更复杂的数据形态，恐怕就得使用对象这种数据结构了。我们可以把 JavaScript 中的对象想象成一个定义的数据结构，使用花括号（{}）构造对象。在花括号之间，是对象的属性和值，两者以冒号分隔。每对属性和值以分号隔开（不包括最后一对属性和值）：

```
var fruit = {
    kind: "grape",
    color: "red",
    quantity: 12,
    tasty: true
};
```

要引用对象中的某个值，可以使用点操作符，然后紧跟着属性名：

```
fruit.kind      // 返回 "grape"
fruit.color     // 返回 "red"
```

```
fruit.quantity  // 返回 12
fruit.tasty     // 返回 true
```

可以把这些值想象为“属于”对象。噢，看哪，这是我的水果（fruit）。“什么水果啊？”你可能会问。水果种类（fruit.kind）不是告诉你它是葡萄（"grape"）了吗。“好吃吗？”当然啦，水果好吃（fruit.tasty）的值是真（true）。

3. 对象加数组

把对象和数组组合起来就可以创建对象的数组，或者数组的对象，或者对象的对象……好吧，反正任何适合你的数据集的数据结构。

假设我们又拿到了更多水果，所以需要扩展库存记录。那么我在外面用方括号，也就是数组，然后在里面用花括号，也就是对象；对象之间呢，当然用逗号分隔：

```
var fruits = [
    {
        kind: "grape",
        color: "red",
        quantity: 12,
        tasty: true
    },
    {
        kind: "kiwi",
        color: "brown",
        quantity: 98,
        tasty: true
    },
    {
        kind: "banana",
        color: "yellow",
        quantity: 0,
        tasty: true
    }
];
```

要读取这种数据结构，只要按照属性去查找值即可。别忘了，[] 表示数组，{} 表示对象。因此，fruits 是数组，首先得用方括号来指定要读取的水果的索引：

```
fruits[1]
```

接下来，每个数组元素就是一个水果对象呗。所以，再加上一个点和属性名就可以了：

```
fruits[1].quantity   // 返回 98
```

下面列出了访问 fruits 对象数组中所有值的方法：

```
fruits[0].kind      ==  "grape"
fruits[0].color     ==  "red"
fruits[0].quantity  ==  12
```

```
fruits[0].tasty     ==  true

fruits[1].kind      ==  "kiwi"
fruits[1].color     ==  "brown"
fruits[1].quantity  ==  98
fruits[1].tasty     ==  true

fruits[2].kind      ==  "banana"
fruits[2].color     ==  "yellow"
fruits[2].quantity  ==  0
fruits[2].tasty     ==  true
```

没错，我们有 `fruits[2].quantity` 个香蕉。

JSON。在使用 D3 的过程中，不可避免地要遇见 JSON（JavaScript Object Notation，JavaScript 对象表示法）。想继续看就看吧，但 JSON 基本就是一个用 JavaScript 对象语法来组织数据的格式。这种语法经过了优化，可以在 JavaScript（显然嘛）和 AJAX 请求中使用。很多基于 Web 的应用程序编程接口（API，Application Programming Interface）都返回 JSON 格式的数据。相对 XML 而言，JSON 更容易在 JavaScript 中被解析，当然 D3 操作它也非常方便。

就这么多了，实际上 JSON 看起来也没什么新东西：

```
{
    "kind": "grape",
    "color": "red",
    "quantity": 12,
    "tasty": true
}
```

唯一的差别就是属性名现在带着双引号，变成了字符串了。

JSON 对象与其他 JavaScript 对象一样，当然也可以保存在变量中：

```
var jsonFruit = {
    "kind": "grape",
    "color": "red",
    "quantity": 12,
    "tasty": true
};
```

GeoJSON。正如 JSON 是基于 JavaScript 对象语法的数据格式，GeoJSON 是基于 JSON 格式的一种格式，专门用于保存地理数据。所有 GeoJSON 对象都是 JSON 对象，所有 JSON 对象都是 JavaScript 对象。

GeoJSON 可以保存地理空间中的点（比如经纬度坐标）或者形状（如直线和多边形），以及其他空间特征。如果你有很多地理数据，为了在 D3 中更方便地利用它

们，最好将其解析成 GeoJSON 格式。

在将来讨论地图的时候，我们还会详细讲解 GeoJSON。现在，只要了解像下面这样简单的 GeoJSON 数据即可：

```
{
    "type": "FeatureCollection",
    "features": [
        {
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [ 150.1282427, -24.471803 ]
            },
            "properties": {
                "type": "town"
            }
        }
    ]
}
```

正好，坐标（`"coordinates"`）数组的两个值分别是经度和纬度。

3.7.4 数学运算符

JavaScript 中执行数值的加、减、乘、除运算分别有相应的数学运算符：

```
+    // 加
-    // 减
*    // 乘
/    // 除
```

例如，在控制台中输入 `8 * 2`，按回车，你会看到 `16`。下面是另外一些例子：

```
1 + 2        // 返回 3
10 - 0.5     // 返回 9.5
33 * 3       // 返回 99
3 / 4        // 返回 0.75
```

3.7.5 比较运算符

要想比较两个值，可以使用下列运算符：

```
==   // 等于
!=   // 不等于
<    // 小于
>    // 大于
<=   // 小于等于
>=   // 大于等于
```

这些运算符都是拿左边的值跟右边的值比较。如果结果为真，返回 true；否则，返回 false。

快试试吧！在控制台里输入以下比较表达式，看看结果如何：

```
3 == 3
3 == 5
3 >= 3
3 >= 2
100 < 2
298 != 298
```

（JavaScript 也提供了 === 和 !== 运算符，这两个相等和不等运算符不会转换数据类型，但本书中不用作这种区分。）

3.7.6　控制结构

在代码需要作出决定或重复执行某些操作时，就要用到控制结构。可选择的控制结构很多，但我们今天只介绍 if 语句和 for 循环。

1. **if()**：只要……

if 语句通过比较运算符来确定某个语句是真是假：

```
if (test) {
    // 如果为真则执行这里的代码
}
```

如果条件测试结果为 true，就运行花括号中的代码。否则，就忽略花括号中的代码，继续向下执行。

```
if (3 < 5) {
    console.log("Eureka! Three is less than five!");
}
```

在前面的例子中，花括号中的代码始终都会执行，因为 3<5 永远为 true。在比较变量或其他可变条件时，if 语句更能派上用场。

```
if (someValue < anotherValue) {
    // 把 someValue 设置成 anotherValue
    // 从而恢复这个世界的平等和秩序
    someValue = anotherValue;
}
```

2. **for()**：现在……

可以使用 for 循环重复执行相同的代码，语法稍有变化：

```
for (initialization; test; update) {
```

```
        // 每次都要执行的代码
    }
```

之所以称它为 for 循环，是因为可以指定循环的次数。首先，运行初始化（initialization ）语句。然后，对测试条件（test ）求值，就像运行一个小 if 语句一样。最后，运行更新语句（update)，再重新对测试条件求值 。

for 循环最常见的用法就是使用一个每次递增 1 的计数器控制循环。（计数器一般都命名为 i，因为很短。）

```
    for (var i = 0; i < 5; i++) {
        console.log(i);  // 把值输出到控制台
    }
```

以上代码会在控制台中循环输出以下结果：

```
    0
    1
    2
    3
    4
```

第一次执行代码时，声明了变量 i，其值初始化为 0。因为 i 小于 5，所以执行花括号内代码，把 i 的当前值（0）输出到控制台。然后，i 加 1（i++ 是 i = i + 1 的简写形式）。好了，现在 i 等于 1，所以测试求值结果仍然为 true，再次执行花括号内代码。但这一次输出到控制台的值是 1。

怎么样，看这种解释代码的文字是不是跟自己亲手执行代码一样有趣啊？这就是人类发明计算机的原因。那我就直接跳到最后吧，在最后一次执行代码之后，i 递增为 5，然后测试求值结果为 false，循环结束。

强调一下，计数器是从 0 而不是 1 开始计数的。换句话说，i 的第一个值是 0。为什么不是 1 呢？反正就是一个惯例啦，不过倒是跟计算机索引数组的值的计数规则恰好一致。

3. 对数组使用for()循环

基于代码的数据可视化如果没有数组和 for() 循环，简直就没法想象了。数组加上 for 循环，正是数据极客的“双杀组合”。（如果你不觉得自己是“数据极客”，那么我得提醒你一下：看看这本书的封面，你在看什么书呢？）

数据把很多数据值组织到了一个方便的地方。而 for() 可以快速“循环”数组中的每个值，对它们执行某种操作，比如把值转换成可见的形式。D3 一般都会替我们执行这种循环，比如使用其魔法般的 data() 方法。注意下面这个例子，它遍历了一

个叫 numbers 的数组中的所有值：

```
var numbers = [ 8, 100, 22, 98, 99, 45 ];

for (var i = 0; i < numbers.length; i++) {
    console.log(numbers[i]);  // 把值输出到控制台
}
```

看到 numbers.length 了吗？ 这是最重要的部分。length 是每个数组都有的属性。对这个例子来说，numbers 包含 6 个值，因此对 numbers.length 求值会得到 6，也就是说，循环会运行 6 次。如果 numbers 包含 10 个值，循环就运行 10 次。如果有 100 万个值……，好，你明白了。这也是计算机最擅长做的事：拿到一组指令，然后反复执行。数据可视化之所以如此吸引人，也是相同的原因。你来设计和编写可视化系统，系统然后就可以随机应变，就算再给它换一批数据也没问题。只要系统的映射规则不变，数据变了也没关系。

3.7.7 函数

函数就是执行具体任务的代码块。

说得具体一点，函数的特殊之处表现在它可以接收参数，可以返回值。圆括号可以调用（执行）函数。如果函数需要参数（输入值），那么就在调用时把参数放到圆括号中。

看到下面这样的代码，你就知道这是在调用函数：

```
calculateGratuity(38);
```

事实上，我们在前面使用 console.log 时，已经在调用函数了：

```
console.log("Look at me. I can do stuff!");
```

但最重要的，还是可以编写自己的函数。在 JavaScript 中有多种方式可以定义函数，下面介绍一种比较简单的方式，本书也会统一采用这种方式：

```
var calculateGratuity = function(bill) {
    return bill * 0.2;
};
```

这里声明了一个新变量，叫 calculateGratuity。然后，没有赋给它一个数值或字符串，而是把一个函数保存在了这个变量中。在函数的圆括号内，我们又声明了另一个变量 bill，只有函数自己可以使用它。bill 就是将来接收输入值的参数。调用函数时，就可以传入相应的参数值，然后函数会把这个值乘以 0.2，返回计算结果。

如果这样调用：

```
calculateGratuity(38);
```

函数就会返回 `7.6`。不错啊，20% 的小费（gratuity）呢！

当然，可以把输出结果保存在另一个变量里，以便将来使用：

```
var tip = calculateGratuity(38);
console.log(tip);  // 在控制台中输出 7.6
```

还有一种匿名函数，跟 `calculateGratuity` 不一样，没有名字[1]。D3 中也使用了大量匿名函数，但我想在遇到的时候再介绍它们。

3.7.8 注释

```
/* JavaScript 支持 CSS 风格的注释 */

// 不过，也可以使用双斜杠
// 双斜杠后面这一行的内容全会被忽略
// 这种单行注释很适合写点简短的说明
// 有时候，也可以放在一行代码的后面：

console.log("Brilliant");  // 把 "Brilliant" 输出到控制台
```

3.7.9 引用脚本文件

脚本可以直接放在 HTML 中，位于一对 `script` 标签之间：

```
<body>
    <script type="text/javascript">
        alert("Hello, world!");
    </script>
</body>
```

或者，保存一个独立的文件中，以 .js 作为扩展名。然后在 HTML（可能像这里一样在 `head` 标签内，也可能在结束的 `body` 标签之前）：

```
<head>
    <title>Page Title</title>
    <script type="text/javascript" src="myscript.js"></script>
</head>
```

注 1：严格来讲，这里展示的定义函数的方式使用了匿名函数表达式，但通过把匿名函数赋值给一个变量，也就相当于给了函数一个名字。如果这里的函数不是匿名的，比如写成 `function calculateGratuity(bill){...}`，那这个函数就叫具名函数表达式。最后一种定义方式是函数声明，即不使用 var，直接写 `function calculateGratuity(bill){...}`。——译者注

3.7.10 JavaScript陷阱

作为一种馈赠，不额外收钱，我再给大家介绍四个 JavaScript 中的陷阱：如果我早知道这些陷阱，可能就不用每次都调试到后半夜了，也会少很多次心急火燎的经历，少分泌一些想让人玩命的皮质醇。在学习本书后面内容的时候，你可能时不时地需要看看这一节。

1. 动态类型

JavaScript 是一种松散类型的语言。换句话说，不必提前声明保存在变量中的数据是什么类型。Java（跟 JavaScript 完全不一样）等其他语言都要求提前声明变量类型，比如 `int`、`float`、`boolean` 或 `String`：

```
// 在 Java 中声明变量
int number = 5;
float value = 12.3467;
boolean active = true;
String text = "Crystal clear";
```

而 JavaScript 则会根据赋给变量的数据，自动推断其类型。（注意，'' 或 "" 表示字符串。我个人喜欢双引号，但有人喜欢单引号。）

```
// 在 JavaScript 中声明变量
var number = 5;
var value = 12.3467;
var active = true;
var text = "Crystal clear";
```

好讨厌哪，那么多 `var`！不过也有方便的地方：不用知道保存什么数据，就可以声明和命名变量。甚至随意改变保存的数据类型，JavaScript 也不会怪你：

```
var value = 100;
value = 99.9999;
value = false;
value = "This can't possibly work.";
value = "Argh, it does work! No errorzzzz!";
```

这里要提醒大家的是，如果你不小心在一个数值变量里保存了一个字符串，后来代码就出现了一些奇怪的问题，希望你自己能好好反省一下。如果你不确定某个变量中保存着什么类型的数据，可以使用 `typeof` 操作符：

```
typeof 67;              // 返回 "number"
var myName = "Scott";
typeof myName;          // 返回 "string"
myName = true;
typeof myName;          // 返回 "boolean"
```

2. 变量提升

在我们印象里，浏览器会从上到下依次执行 JavaScript 代码。但有时候也不一定！比如，下面这些代码，你觉得变量 i 什么时候有定义？

```
var numLoops = 100;
for (var i = 0; i < numLoops; i++) {
    console.log(i);
}
```

在其他很多语言中，i 会到 for 循环开始时才被声明，这符合我们的预期。但由于存在一种叫做变量提升的机制，JavaScript 中的变量声明会被提升到函数上下文的顶部。对于前面的例子来说，i 实际上在 for 循环开始之前就有了定义。下面的代码跟上面的代码是等价的：

```
var numLoops = 100;
var i;
for (i = 0; i < numLoops; i++) {
    console.log(i);
}
```

如果变量名字有冲突，变量提升可能就会带来问题。因为你本以为某个变量到后面才会有定义，不成想它早在代码一开始执行时就有定义了。

3. 函数级作用域

在编程领域，变量作用域这个概念用于区分在哪个环境下可以访问哪些变量。一般来说，在任何地方都能访问任何值是不正确的。因为这样发生冲突的可能性太高，或者不知道在哪儿意外改了某个值，都会让你急得疯掉。

很多语言都有块级作用域，也就是在当前“块”中，变量才有定义。这里的“块”，通常就是花括号。在块级作用域下，前面例子中的变量 i 将只存在于 for 循环中。如果想在循环外部读取或修改 i 的值，都会失败。这样其实挺好，因为你知道循环中的变量不会跟循环外部的其他变量有冲突。

但是，JavaScript 中的变量只能限制在函数中，即在函数（而不是任何块）中声明的变量只能在函数内部访问。

假如你使用过其他语言，那么对这一点必须格外注意。关键要记住：要想封闭某个值，就得把它们放到函数里。

4. 全局命名空间

说到变量冲突，请帮我个忙：打开任意网页，调出 JavaScript 控制台，输入 window，然后单击那个灰色三角形，看看里面都有什么。

我知道你就像进入了迷宫一样，而这正是所谓的“全局命名空间”，真是“百闻不如一见”（参见图 3-10）。

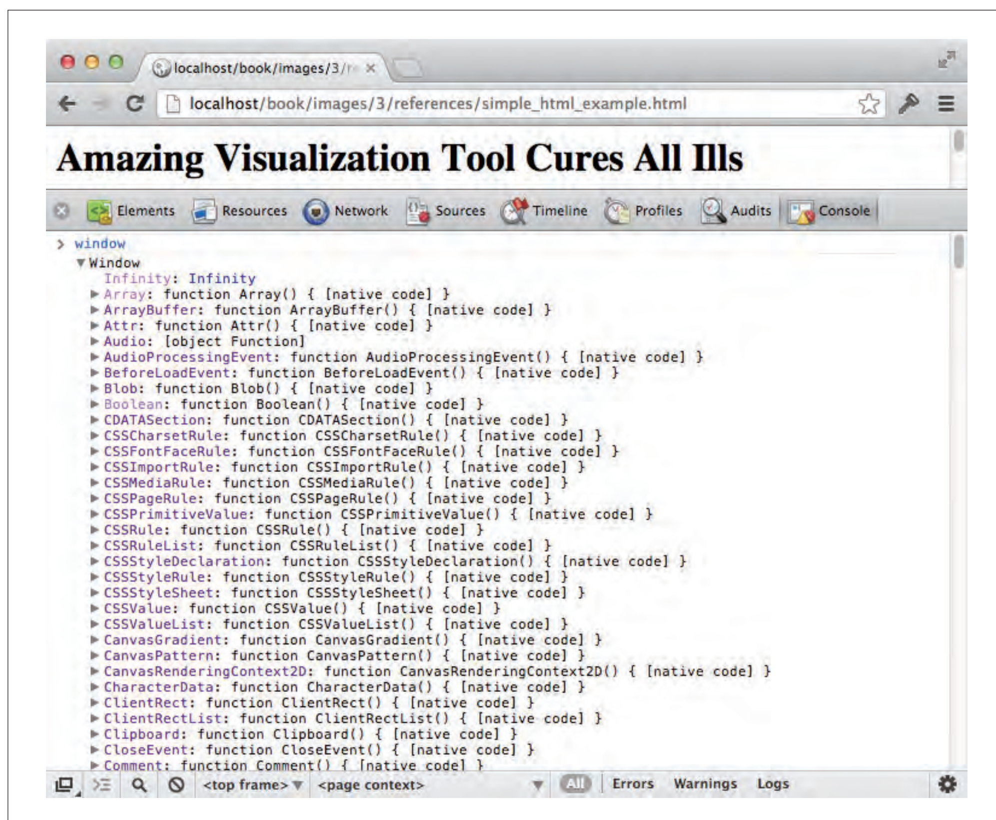


图 3-10：全局命名空间

`window` 在浏览器中是一个顶级对象，包含所有 JavaScript 中能直接访问到的对象。你看到的所有这些对象和值都包含在全局作用域中。换句话说，如果你每次都在全局作用域中声明新变量，那这个变量就会被加到 `window` 对象下面。正像一些 JavaScript 界敢于仗义直言的人所说：“你这是在污染全局命名空间。”（奇怪的是，很多在论坛里义正词严说这种话的人，在现实当中都很平易近人。）

比如，我们在控制台中输入：**`var zebras = "amazing"`**。

然后，再输入 `window`，点开灰色小三角，一直向下滚动，滚动到最底部，就会看到 `zebras`，如图 3-11 所示。

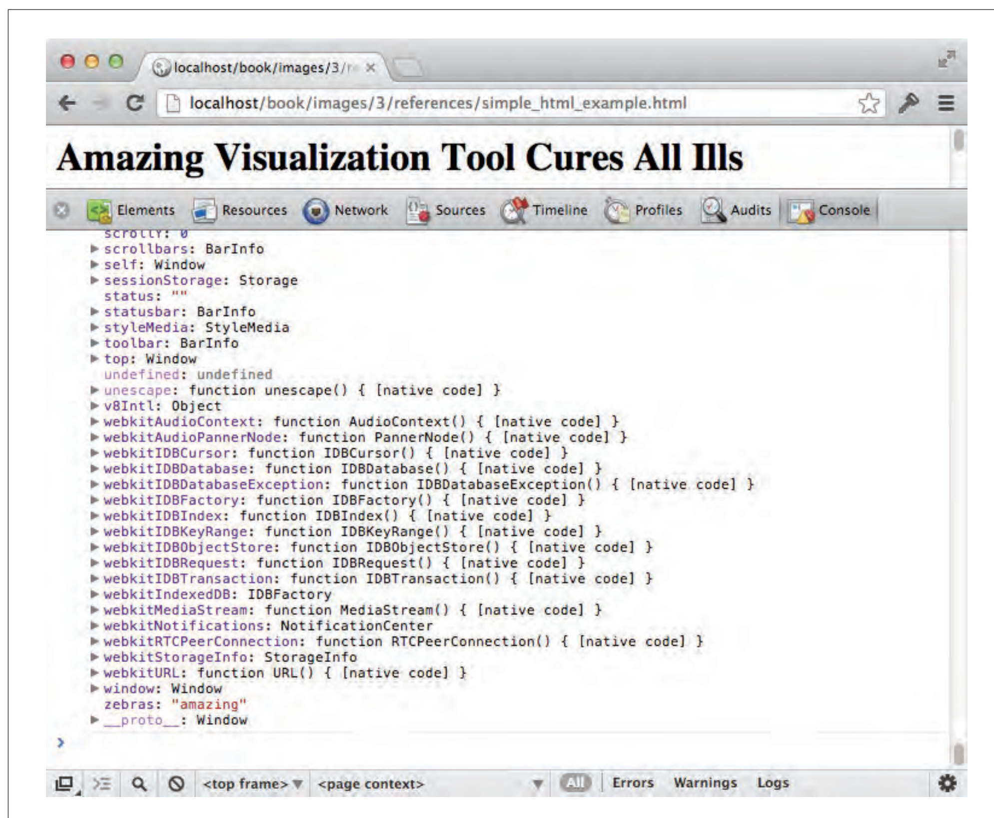


图 3-11：全局作用域中有了 `zebras`

（让人不解的是，在 JavaScript 中不使用标准的 `var` 也可以定义新变量。因此，只要输入 **`zebras = "amazing"`** 也会得到与前面相同的结果。）

给 `window` 增加几个值有什么大问题吗？刚开始的时候，啥事儿也没有。但随着你的项目越来越复杂，特别是在需要联合使用非 D3 的 JavaScript 代码时（如 jQuery、Facebook 的“Like”按钮，或谷歌的分析跟踪代码），说不定就会发生命名冲突。比如，你在自己的项目里使用了变量 `zebras`，而 zebraTracker.js 恰好也使用了一个同名变量！结果肯定是一团糟。至少，一些不太规范的数据，很有可能导致你的代码发生异常或错误。

解决这个问题有两个简单的办法（说明一下，不到后面你不用担心这一点）。

- 只在函数里面声明变量。虽然有时候也不是绝对可行，但函数级作用域可以防止其本地变量跟其他变量发生冲突。
- 只声明一个全局对象，然后把你本来想作为全局变量的值都作为这个对象的属性。

比如：

```
var Vis = {}; // 声明空的全局对象
Vis.zebras = "still pretty amazing";
Vis.monkeys = "too funny LOL";
Vis.fish = "you know, not bad";
```

这样，所有变量就都被“关在笼子里了”（在这里就是全局对象 `Vis` 里），因此不会再污染全局命名空间。当然，不一定非得叫 `Vis`，叫什么随你便——不过叫 `Menagerie`（动物园）输入起来就太麻烦了。无论如何，如果再有冲突发生，那肯定是其他脚本里也恰好有一个全局对象，而且也起了个相同的名字（`Vis`）。但这种事儿发生的概率就低得多了。

3.8 SVG

D3 最适合用来生成和操作 SVG（Scalable Vector Graphics，可伸缩矢量图形）。通过 `div` 和其他原生 HTML 元素也没问题，但总是略显笨重，而且会遭遇浏览器间不一致的问题。使用 SVG 更加可靠，图形效果更一致，而且速度也更快。

图 3-12 展示了一个小圆形，对应的 SVG 代码在下面。

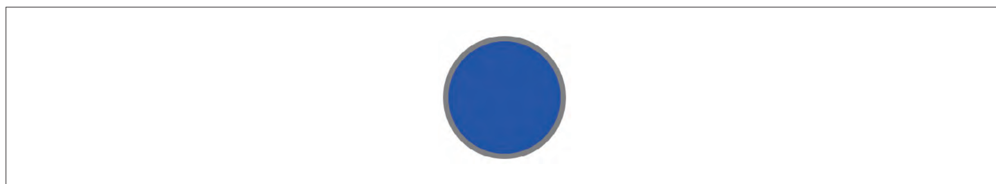


图 3-12：小 SVG 圆形

```
<svg width="50" height="50">
    <circle cx="25" cy="25" r="22" fill="blue" stroke="gray" stroke-width="2"/>
</svg>
```

可以使用 Illustrator 等矢量生成软件来生成 SVG 文件，但我们要学习的是如何通过代码来生成它们。

3.8.1 SVG元素

SVG 是一种文本格式。每个 SVG 图形都是使用与 HTML 类似的标记定义的，而且 SVG 代码可以直接包含在 HTML 文档中，或者动态插入到 DOM 中。除了 IE8 及之前版本之外，所有浏览器都支持 SVG。SVG 也是一种 XML 语言，所以那些不包含结束标签的元素，一定要自关闭。比如：

```
<element></element> <!-- 带关闭标签 -->
<element/>          <!-- 自关闭元素 -->
```

在学习绘图之前，首先得创建 SVG 元素。可以把 SVG 元素想象成一个可以在上面作画的画布。（从这个意义上说，SVG 与 HTML 中的 canvas 元素相像。）至少要给新 SVG 元素指定 width 和 height 值。如果不指定，SVG 就会像典型的块级 HTML 元素一样，贪婪地在包含它的元素内尽可能地扩大地盘。

```
<svg width="500" height="50">
</svg>
```

注意，像素是默认的度量单位，因此可以只指定 500 和 50，而不是 500px 和 50px。当然也可以明确地给出单位，除了像素之外，支持的其他单位还包括 em、pt、in、cm 和 mm。

3.8.2　简单的图形

在 svg 标签中可以嵌入很多可见的元素，包括 rect、circle、ellipse、line、text 和 path。

如果你熟悉计算机图形编程，那肯定知道基于像素的坐标系统的原点（即 0,0 点）位于画布的左上角。增大 x 的值，图形会向右移动；增大 y 值，图形会向下移动（参见图 3-13）。

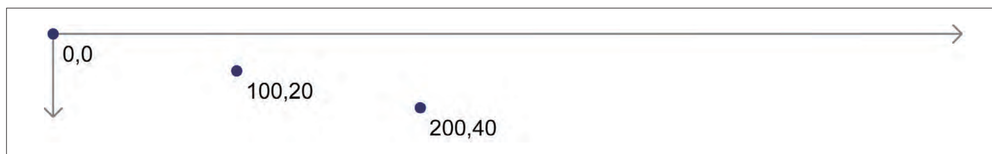


图 3-13：SVG 坐标系统

rect 用于绘制矩形。x 和 y 用于指定矩形左上角的坐标，而 width 和 height 用于指定其大小。下面这个矩形会填满 SVG 元素的整个空间，如图 3-14 所示。

```
<rect x="0" y="0" width="500" height="50"/>
```



图 3-14：SVG 矩形

circle 用于绘制圆形。cx 和 cy 用于指定圆心坐标，而 r 用于指定半径。下面这

个圆形在 500 像素宽的 SVG 画布上居中，因为其 cx（center-x，即圆心 x 坐标）值等于 250，参见图 3-15。

```
<circle cx="250" cy="25" r="25"/>
```

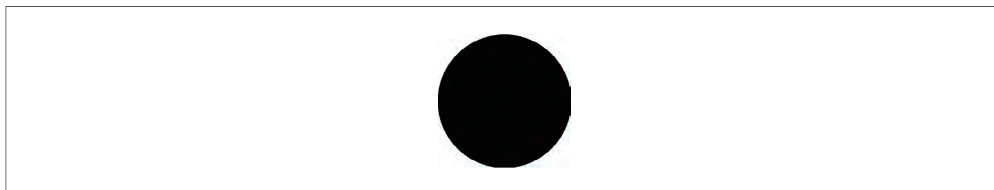


图 3-15：SVG 圆形

ellipse 与 circle 类似，只不过每个轴要分别指定半径。因此，半径属性不再是 r，而是 rx 和 ry，结果如图 3-16 所示。

```
<ellipse cx="250" cy="25" rx="100" ry="25"/>
```



图 3-16：SVG 椭圆形

line 用于绘制直线，如图 3-17 所示。x1 和 y1 用于指定起点坐标，x2 和 y2 用于指定终点坐标。另外，必须用 stroke 指定直线的颜色才能看得见。

```
<line x1="0" y1="0" x2="500" y2="50" stroke="black"/>
```

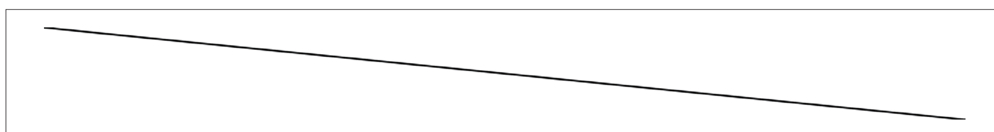


图 3-17：SVG 直线

text 用于绘制文本。x 用于指定文本左上角的位置，y 用于指定字体的基线位置。（基线是印刷术语，指文本中一条不可见的线，所有字母都以之为基准对齐。“p”和“y”这样的字母会伸到基线以下，伸出的部分叫下伸部分。）以下代码会得到如图 3-18 所示的结果。

```
<text x="250" y="25">Easy-peasy</text>
```

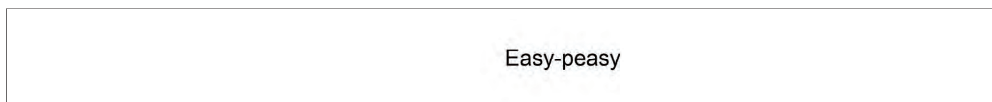


图 3-18：SVG 文本

SVG 文本会继承 CSS 为父元素指定的字体样式，除非另有指定。（关于给文本添加样式，我们稍后再介绍。）要覆盖继承的样式，可以像下面这样，结果如图 3-19 所示。

```
<text x="250" y="25" font-family="serif" font-size="25"
 fill="gray">Easy-peasy</text>
```



图 3-19：指定 SVG 文本的样式

还要注意一点，要防止任何可见的元素跑到 SVG 画布外面，否则会被切掉。特别是对于文本，英文字母的下伸部分一不小心就会跑出去。图 3-20 展示了这样一种情况，因为基线（y）被设置为 50，与 SVG 画布高度一样。

```
<text x="250" y="50" font-family="serif" font-size="25"
 fill="gray">Easy-peasy</text>
```



图 3-20：SVG 文本跑出去了

path 用于绘制前面图形之外的其他复杂图形（如地图上的国境线），后面我们会单独介绍。本节只介绍简单的图形。

3.8.3 为SVG元素添加样式

SVG 的默认样式是黑色填充，没有描边。如果你想添加其他样式，必须自己给元素添加。常见的 SVG 属性有下面这些。

- fill
  颜色值。与使用 CSS 一样，可以使用颜色名、十六进制值，或 RGB、RGBA 值。

- stroke
  颜色值。

- `stroke-width`

  带单位的数值（通常单位是像素）。

- `opacity`

  0.0（完全透明）到 1.0（完全不透明）之间的数值。

对于文本，也可以使用下面这些属性，使用方式与 CSS 中类似。

- `font-family`
- `font-size`

另一个与 CSS 类似的地方，是可以通过两种方式给 SVG 元素应用样式：作为元素属性直接应用（行内）或使用 CSS 样式规则。

以下是通过属性方式给圆形应用样式的例子，结果如图 3-21 所示。

```
<circle cx="25" cy="25" r="22"
 fill="yellow" stroke="orange" stroke-width="5"/>
```



图 3-21：SVG 圆形

另外，也可以删除所有样式属性，通过 `article` 类来为它定义样式（就像给其他 HTML 元素应用样式一样）：

```
<circle cx="25" cy="25" r="22" class="pumpkin"/>
```

在样式规则中，使用同样的 `fill`、`stroke` 和 `stroke-width` 属性定义这个类的样式：

```
.pumpkin {
    fill: yellow;
    stroke: orange;
    stroke-width: 5;
}
```

CSS 方式有以下两个明显的好处：

- 可以只写一次样式，然后把它们应用给多个元素；
- CSS 代码比行内属性容易看懂。

- 正因为如此，CSS 方式更便于维护，也可以更快地修改设计。

不过，有时候通过 CSS 给 SVG 应用样式会令人不太放心。因为 fill、stroke 和 stroke-width 都不是 CSS 属性。（最相近的 CSS 属性是 background-color 和 border。）而我们只是在利用 CSS 选择符来应用 SVG 专有的属性。为了方便区分哪些样式规则是应用给 SVG 的，可以考虑在选择符中增加 svg 标记：

```
svg .pumpkin {
    /* ... */
 }
```

3.8.4 分层与绘制顺序

SVG 中没有“层”和深度的概念，也不支持 CSS 的 z-index 属性，因此属性只能在二维画布表面上排布。

不过，如果要绘制多个图形，它们是会重叠的：

```
<rect x="0" y="0" width="30" height="30" fill="purple"/>
<rect x="20" y="5" width="30" height="30" fill="blue"/>
<rect x="40" y="10" width="30" height="30" fill="green"/>
<rect x="60" y="15" width="30" height="30" fill="yellow"/>
<rect x="80" y="20" width="30" height="30" fill="red"/>
```

代码中元素出现的顺序决定了它们的深度次序。在图 3-22 中，紫色（purple）方形在代码中出现得最早，因此浏览器首先绘制它。然后，在它的“上面”绘制蓝色（blue）方形。接着依次是绿色、黄色和红色的方形。



图 3-22：SVG 元素重叠

可以把浏览器绘制 SVG 图形想象成在画布上作画。后涂的油漆会使先涂的油漆变模糊，因此出现在了“前面”。

在需要同时绘制多个视觉元素，但又不想让它们相互遮盖时，绘制的顺序至关重要。比如，我们需要给散点图画上数轴和数值标签，那么可以在 SVG 中最后再添加数轴和标签，这样它们就可以出现在其他元素前面。

3.8.5 透明度

在元素相互遮挡，但又不能完全遮住先绘制的元素时，或者想弱化某些元素而突出显示其他元素时，透明度效果扮演着重要角色，如图 3-23 所示。

有两种应用透明度的方式：或者使用带透明通道的 RGB 值，或者设置 `opacity`（不透明度）值。

可以在为 `fill` 或 `stroke` 属性指定颜色的时候使用 `rgba()`。`rgba()` 接受 0 到 255（包含及）之间的三个值，分别代表红、绿、蓝，还接受第四个 0.0 到 1.0（包含及）之间的透明度值。

```
<circle cx="25" cy="25" r="20" fill="rgba(128, 0, 128, 1.0)"/>
<circle cx="50" cy="25" r="20" fill="rgba(0, 0, 255, 0.75)"/>
<circle cx="75" cy="25" r="20" fill="rgba(0, 255, 0, 0.5)"/>
<circle cx="100" cy="25" r="20" fill="rgba(255, 255, 0, 0.25)"/>
<circle cx="125" cy="25" r="20" fill="rgba(255, 0, 0, 0.1)"/>
```



图 3-23：RGBA SVG 图形

使用 `rgba()` 可以分别为填充（`fill`）和描边（`stroke`）颜色应用透明度。下面几个圆形填充都是 75% 不透明，而它们的描边都只有 25% 不透明：

```
<circle cx="25" cy="25" r="20" fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10"/>
<circle cx="75" cy="25" r="20" fill="rgba(0, 255, 0, 0.75)"
        stroke="rgba(0, 0, 255, 0.25)" stroke-width="10"/>
<circle cx="125" cy="25" r="20" fill="rgba(255, 255, 0, 0.75)"
        stroke="rgba(255, 0, 0, 0.25)" stroke-width="10"/>
```

应用透明度之后，可以得到图 3-24 所示的结果。描边与每个圆形的边缘对齐，既不完全位于圆形内部，也不完全位于圆形外部，而是一半在内一半在外。在应用了透明度之后，每个 10 像素的描边看起来倒像是两个 5 像素的描边一样。

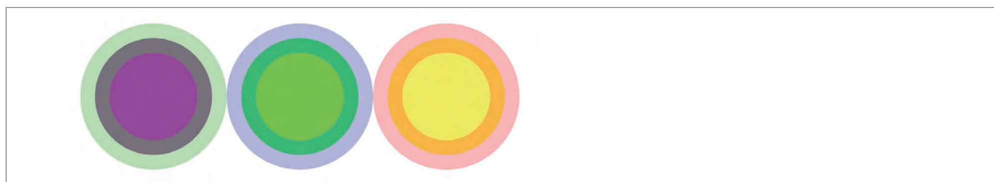


图 3-24：RGA SVG 图形中的填充和描边

要给整个元素应用透明度，可以设置 opacity 属性。图 3-25 展示了几个完全不透明的圆形。

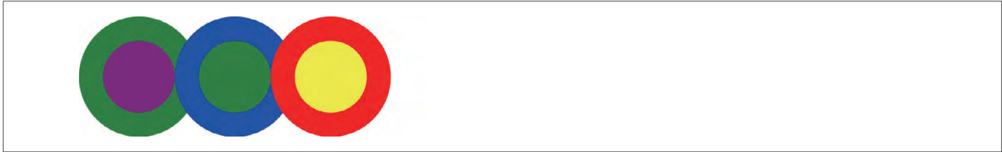


图 3-25：完全不透明的圆形

图 3-26 展示了同样的圆形但设置了 opacity 值的效果：

```
<circle cx="25" cy="25" r="20" fill="purple" stroke="green" stroke-width="10"
        opacity="0.9"/>
<circle cx="65" cy="25" r="20" fill="green" stroke="blue" stroke-width="10"
        opacity="0.5"/>
<circle cx="105" cy="25" r="20" fill="yellow" stroke="red" stroke-width="10"
        opacity="0.1"/>
```

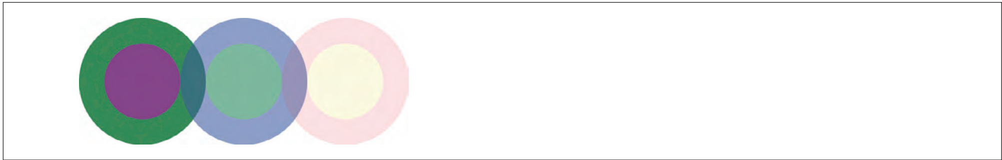


图 3-26：部分透明的圆形

也可以对已经使用 rgba() 设置了颜色的元素应用 opacity 属性。这时候，透明度值会相乘。图 3-27 所示的三个圆形的 fill 和 stroke 分别使用了相同的 RGBA 值。而且除第一个圆形外，另外两个都设置了 opacity 值：

```
<circle cx="25" cy="25" r="20" fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10"/>
<circle cx="65" cy="25" r="20" fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10" opacity="0.5"/>
<circle cx="105" cy="25" r="20" fill="rgba(128, 0, 128, 0.75)"
        stroke="rgba(0, 255, 0, 0.25)" stroke-width="10" opacity="0.2"/>
```

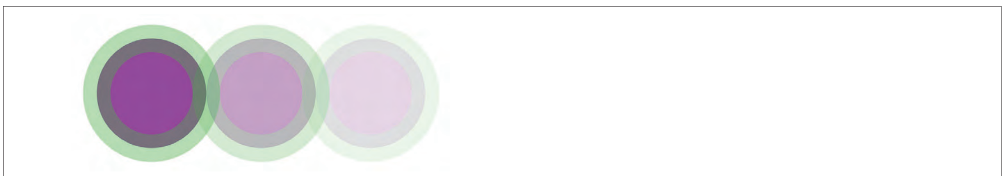


图 3-27：RGBA 与 opacity 相乘的效果

以第三个圆形为例，其 opacity 值是 0.2（即 20%）。而其紫色的 fill 值中的透

明通道值是 0.75（即 75%）。结果，紫色区域最终的透明度就是 0.2 × 0.75 = 0.15（即 15%）。

3.9 关于兼容性

旧版本浏览器不支持 SVG。正常情况下，IE8 及更早版本不会显示 SVG 图形。有时候确实有点讨厌，因为本该华丽无比的图形成了一片空白。而且，D3 本身也不支持旧版本浏览器（当然，主要是 IE8 及更早版本）。总之，两者简直形同水火，更谈不上兼容性了。



在配合使用 Aight 兼容库（https://github.com/shawnbot/aight）的情况下，D3 可以在 IE8 中运行，但更早版本的 IE 就不行了。r2d3（https://github.com/mhemesath/r2d3/）是 D3 整合 Raphaël（http://raphaeljs.com/）的图形绘制功能后一个版本，可以尽可能兼容 IE7。

当然，最好的办法还是鼓励人们升级，或者选择最近几年才流行的浏览器。使用现代浏览器的人越多，他们的上网体验也就越好，而我们潜在的用户也就越广。

换句话说，告诉用户为什么这个功能不能用是对用户的尊重。建议使用 Modernizr（http://modernizr.com/）或类似的 JavaScript 工具检测浏览器是否支持 SVG。如果支持，就加载 D3 代码，并正常处理；否则，就显示静态的非交互性的图表，同时附上一段说明，解释为什么需要较新版本的浏览器。（措词一定要委婉，同时提供 Chrome 和 Firefox 下载页面的链接。我以前也提供下载 Safari 的链接，但苹果后来将其整合到了 Mac OS 中，不再提供单独下载了。）

举个例子，可以使用 `Modernizr.load()` 检查 Modernizr 测试的结果。如果测试通过（比如检测到浏览器支持 SVG），则告诉 Modernizr 加载 D3 和你自己的 JavaScript 代码。我一般会在文档的 `<head>` 标签中放如下代码：

```
<script src="js/modernizr.js"></script>

<script type="text/javascript">
        Modernizr.load({
                test: Modernizr.svg && Modernizr.inlinesvg,
                yep : [ 'js/d3.v3.min.js',
                        'js/script.js' ]
        });
</script>
```

首先加载 Modernizr，执行检测，然后只在测试成功的情况下才加载其他代码（即 `yep` 部分）。至于检测失败后显示什么，可以使用 CSS 或其他 JavaScript 脚本，也可

以显示一幅静态图表和鼓励用户尝试新浏览器的几句话。要了解如何通过 Modernizr 实现这些，可以参考它的文档，地址：http://modernizr.com/docs/#load。

caniuse.com 也是一个非常难得的资源，通过它可以查询浏览器支持哪些特性，以及哪些浏览器支持 SVG（http://caniuse.com/#search=svg）。

有人也尝试让 D3 在老版本浏览器中运行，有时候需要混合使用 Raphaël 在 `canvas` 上渲染，或者使用其他偏门技术。虽然技术上不是不可能，但我不推荐这么做。

第 4 章

安装D3

安装 D3 是非常简单的，只要下载它的最新版本，新建一个空页面来写代码，最后再设置一下本地服务器就好了。

4.1 下载D3

先创建一个新文件夹，保存项目文件。文件夹的名字随便起，建议叫 project-folder。

在这个文件夹里，最好再建一个子文件夹叫 d3。然后把下载的 D3 的最新版本保存在这个子文件夹里。因为 D3 下载下来是个 ZIP 文件，所以要把它解压缩。在写作本书时，D3 的最新版本是 3.0.6（http://d3js.org/d3.v3.zip）。

D3 还提供了一个“瘦身”版本 d3.v3.min.js（http://d3js.org/d3.v3.min.js），由于去掉了空格，体积更小，加载速度也更快。虽然功能都一样，但在开发项目时最好还是用正常的版本（为了调试方便），等项目可以对外公开发布时再换成缩小版（为了减少浏览器加载的时间）。到底用哪一个还是你说了算，反正本书会使用标准版。

另外，还可以下载整个 D3 代码库（https://github.com/mbostock/d3/zipball/master），其中不光包含 JavaScript 文件，还有各种源代码文件。可以先浏览一下代码库（https://github.com/mbostock/d3），或干脆直接把整个压缩文件下载下来。当然，全都下载完了，还要把其中的 d3.v3.js 复制到 project-folder/d3/ 里。

4.2 引用D3

接下来，在项目文件夹里创建一个 HTML 页面，命名为 index.html。记住，HTML 文档就是普通的纯文本文件。TextEdit 和 Notepad 等文本编辑器都可以，但最好是找一个专门为编码设计的，比如 Coda、Espresso 或 Sublime Text（还有很多很多）。

如果在保存时编辑器提供了文件编码选项，选择 Unicode (UTF-8)。

现在，文件夹的结构如下所示：

```
project-folder/
    d3/
        d3.v3.js
        d3.v3.min.js (optional)
    index.html
```

把下面的代码粘贴到新建的 HTML 文件中：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>D3 Page Template</title>
        <script type="text/javascript" src="d3/d3.v3.js"></script>
    </head>
    <body>
        <script type="text/javascript">
            // 你的 D3 代码
        </script>
    </body>
</html>
```

如果这点事都不想做，可以直接下载本书示例代码文件（参见第 1 章），找到 01_empty_page_template.html，其中的代码与上面的完全相同（但 `src` 路径里 D3 的版本号可能不一样，你得自己改成正确的）。

关于这个模板，有以下几点说明。

- `meta` 标签注明文件的编码为 `utf-8`，这是确保浏览器正确解析 D3 的功能和数据的关键；
- 第一个 `script` 标签设定了对 d3.v3.js 的引用。如果你想使用缩小版或加载保存在其他地方的 D3 文件，就需要修改这里的文件路径。
- 第二个 `script` 标签在 `body` 中，是你编写代码的地方。相信我，你的代码一会很漂亮。

完成！ D3 模板文件和文件夹都各就各位了。如果你以后还想创建其他项目，可以

把这个模板项目完整地复制过去。

4.3 配置Web服务器

有时候，可以直接在浏览器中查看本地 HTML 文件。但出于安全方面的考虑，某些浏览器可能会限制 JavaScript 加载本地文件。这样的话，如果 D3 代码需要加载外部的数据文件（如 CSV 或 JSON），那就会出问题了。但这不是 D3 的问题，而是浏览器在阻止加载来自第三方不受信任站点的脚本和其他外部文件。

为了解决这个问题，更可靠的方案是把页面放到 Web 服务器上。建议大家在本机（也就是你眼前这台电脑）上安装一个 Web 服务器软件，这样比使用远程 Web 服务器更方便也更快捷。本地计算机作为服务器，托管并向自己提供文件好像有点奇怪。你可以这么想啊，浏览器和 Web 服务器是两个程序，前者向后者请求文件，后者为前者提供服务。

实际上，在本机上安装 Web 服务器非常容易。以下介绍几种方式。

4.3.1 基于Python的文本终端方案

如果你使用的是 Mac OS X 或 Linux，那已经安装了 Python。只要你熟悉在文本终端中输入合集，就可以直接运行一个基于 Python 的迷你服务器。这绝对是最简便的方法。（如果你使用 Windows，得先安装 Python 才行。）

要使用 Python，打开系统中的终端窗口。在 Mac 上，打开 Terminal 程序，一般是在 Utilities 文件夹下。或者，直接在 Spotlight（屏幕右上角的放大镜菜单）中直接输入 Terminal。Linux 用户天生知道怎么打开终端窗口，因此我就不浪费笔墨了。

运行 Python 服务器的步骤如下。

1. 打开一个新的终端窗口。
2. 在命令行中找到你想要公开的文件夹。假如你的项目文件夹在 Mac 的 Desktop 文件夹里，可以输入：`cd ~/Desktop/project-folder`。
3. 再输入 `python -m SimpleHTTPServer 8888 &`。

（以上命令在 Python 2.x 中可以使用，但 Python 3.0 及更新版本已经去掉了 SimpleHTTPServer。对 Python 3.x，只要把命令中的 `SimpleHTTPServer` 替换成 `http.server` 即可。）

这样就能在 8888 端口激活服务器。切换到浏览器，访问以下 URL：http://localhost:8888/。没错，不用输入 www.something.com 之类的网址，只要使用

localhost 即可，它的意思是让浏览器请求位于本机上的页面。

然后就会看到一个空的“D3 Page Template”页面。因为页面的主体是空的，所以在浏览器中什么也看不到。选择“查看源代码”，就能看到 HTML 模板页面的内容。

4.3.2　MAMP、WAMP和LAMP

这个方案要费点时间，如果你不喜欢终端，而是希望通过拖放来安装程序，可以试一试。

标题中的 AMP 代表 Apache（Web 服务器软件）、MySQL（流行的数据库软件）和 PHP（流行的服务器端脚本语言）。我们是主要想用 Apache，那才是 Web 服务器软件，不过另外两个已经跟它绑定了，而且相处融洽。

在 Mac 上，可以下载并安装 MAMP（http://mamp.info/en/）或 XAMPP for Mac（http://www.apachefriends.org/zh_cn/xampp-macosx.html）。

这两个软件对应的 Windows 版分别是 WampServer（http://www.wampserver.com/en/）和 XAMPP for Windows（http://www.apachefriends.org/zh_cn/xampp-windows.html）。

如果你使用 Linux，那么这些程序可能都已经安装好。不过，你还是可以再下载 XAMPP for Linux（http://www.apachefriends.org/zh_cn/xampp-linux.html）。

以上这些程序的安装过程各有不同，建议安装时注意看说明。（我发现最容易安装的是 MAMP。）

安装时，每个程序都会指定一个文件夹作为服务器的目录，以便只公开该目录下的文件。安装完了，你需要找到这个目录，把 D3 的 project-folder 文件夹复制过去。

安装好本地服务器，接下来就可以通过浏览器（当然是同一台机器上的浏览器）查看其目录下的页面了。在浏览器地址栏输入：http://localhost/，后面跟不跟端口号，要看你的 AMP。如果 AMP 配置成通过 8888 端口提供服务，那在地址栏中应该输入：http://localhost:8888/。

如果服务器端口号是 8888，你的项目文件夹是 project-folder，那你的 D3 模板页面地址应该是：http://localhost:8888/project-folder/。

4.3.3　快开始吧

准备就绪了？好，下一章就开始讲数据。

第 5 章
数据

数据是一个非常宽泛的概念，其宽泛程度仅次于无所不包的信息。什么是数据?（什么不是数据？）你手里有什么数据，D3 需要什么数据?

好吧，宽泛地说，数据就是结构化的信息，反映某些事实。

在可视化编程的语境下，数据保存在数字化文件中，一般是文本格式或二进制格式。当然，并不是只有文本内容才算数据，那些表示图像、音频、视频、数据库、流、模型、文档等一切的比特和字节也是数据。

然而，从 D3 和浏览器可视化的角度来说，我们只讨论文本数据。换句话说，就是那些可以表现为数值或字符串的东西。如果你可以把数据保存到 .txt 纯文本文件，或者 .csv 逗号分隔值文件，或者 .json JSON 文档中，那么 D3 就可以使用它。

不管什么数据，如果不附着在什么东西上，那么就很难使用和可视化。用 D3 的术语来说，数据必须绑定到页面中的元素上。所以，本章我们先讨论怎么创建新的页面元素，然后再介绍怎么把数据绑定到这些元素。

5.1 生成页面元素

通常，在使用 D3 创建新 DOM 元素时，新元素可以是圆形、矩形，或者其他可以表现数据的图形。但为了大家好理解，我们先从创建简单的 p 元素开始。

为此，首先要根据上一章的 HTML 模板创建一个新文档。可以在示例代码中找到

01_empty_page_template.html，其代码如下所示。（眼光犀利的读者会注意到，src 属性中的文件路径不一样，那是为了适应示例代码的目录结构。不太明白也没关系。）

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>D3 Page Template</title>
        <script type="text/javascript" src="../d3/d3.v3.js"></script>
    </head>
    <body>
        <script type="text/javascript">
            // 你的 D3 代码
        </script>
    </body>
</html>
```

在浏览器中打开这个页面。一定要通过本地 Web 服务器来访问这个文件，第 4 章介绍过。因此，浏览器中的 URL 应该大致像下面这样：

```
http://localhost:8888/d3-book/chapter_05/01_empty_page_template.html
```

如果不是通过 Web 服务器访问这个页面，那么 URL 的开头就不是 http://，而是 file://。再确认一下，保证 URL 长得不是这样：

```
file:///.../d3-book/chapter_05/01_empty_page_template.html
```

好，浏览器加载完页面后，打开 Web 检查器。（关于 Web 检查器，可以参考 3.4 节。）现在看到的只是一个空页面，DOM 内容如图 5-1 所示。

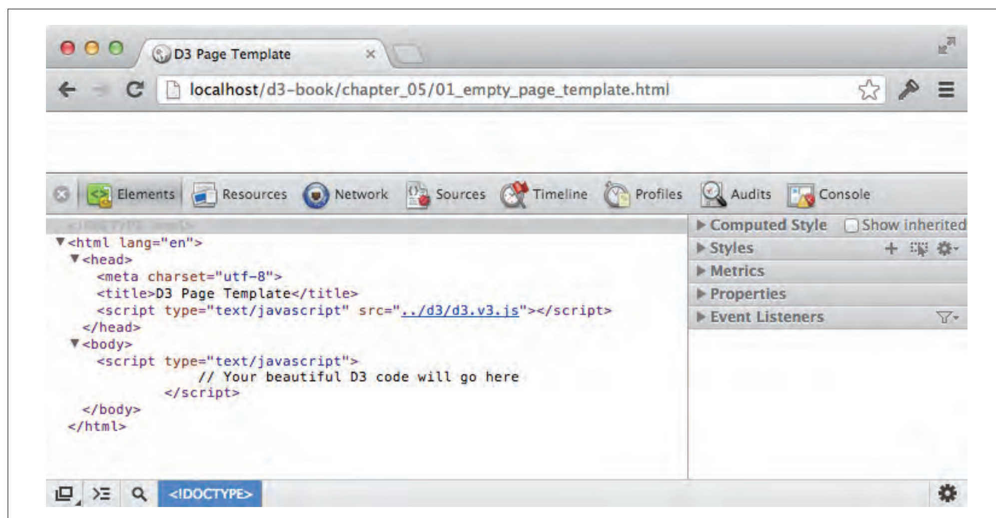


图 5-1：Web 检查器

回到文本编辑器，把 script 标签中的注释替换成以下代码：

```
d3.select("body").append("p").text("New paragraph!");
```

保存，然后刷新浏览器。哇喔，原来的空白页面上多了一行字，而 Web 检查器也变成了图 5-2 所示。

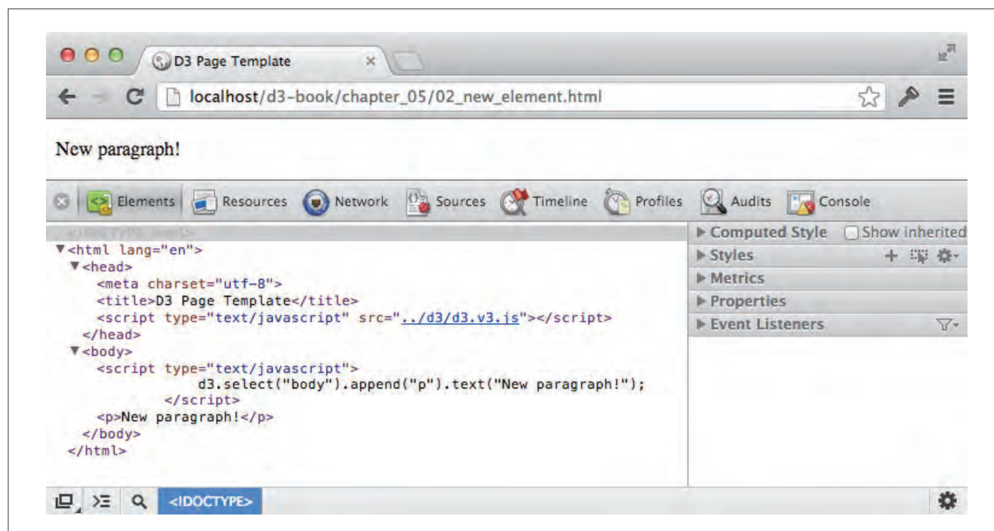


图 5-2：添加代码后的 Web 检查器

看到不同了吗？现在的 DOM 中多了一个段落元素，这个元素就是代码动态生成的！只有这一个元素不算什么，呆会儿你会看到使用类似的技术可以动态生成几十上百个元素，而每个元素都对应着你数据集中的一个值。

接下来我们解释一下到底发生了什么。（没照着做的读者可以打开 02_new_element.html。）为了理解第一行 D3 代码，必须先认识一位新朋友：*连缀语法*。

5.1.1 连缀方法

D3 聪明地利用了一种叫*连缀语法*的技术，用过 jQuery 的朋友都知道。通过用句点把方法“连缀”在一起，一行代码就能执行多个操作。对于这种简单便捷的语法，最关键是要理解其原理，以避免你将来调试时的麻烦。

噢对了，函数和方法是同一个概念的两种说法，它们都是能够接收参数作为输入，然后执行某种操作，最后返回信息作为输出的一段代码。

乍一看，下面这行代码

```
d3.select("body").append("p").text("New paragraph!");
```

好像非常乱，特别是没有编程经验的人，会有这种感觉。所以首先得知道 JavaScript 跟 HTML 一样，都不关心空格和换行。为此，可以把每个方法调用各放在一行上：

```
d3.select("body")
    .append("p")
    .text("New paragraph!");
```

我是推荐你把每个方法调用都分别写在一行上，但某些程序员可能有自己的编码习惯。缩进多少、是否换行或使用空格还是制表符，全都取决于你。

5.1.2 各个击破

好了，下面就来逐个分析上面那串代码。

- `d3`
  引用 D3 对象，从而能够调用其方法。D3 探险就此开始了。

- `.select("body")`
  向 `select()` 方法传入一个 CSS 选择符作为输入，它就会返回一个对 DOM 中匹配的第一个元素的引用。（如果你想取得多个元素，可以使用 `selectAll()` 方法。）这里只是取得文档的 `body` 元素，然后把对它的引用交给调用链中的下一个方法。

- `.append("p")`
  `append()` 会创建一个你指定的新 DOM 元素，然后将它追加到调用它的元素末尾（作为最后一个子元素）。这里创建了一个 `p` 元素，因为调用该方法的是 `body`，所以结果就是在 `body` 元素内部追加一个 `p` 元素。最后，`append()` 把刚刚创建的新元素的引用交给下一个方法。

- `.text("New paragraph!")`
  `text()` 接受一个字符串，把它插入到当前元素的开始和结束标签之间。因为上一个方法传递过来一个 `p` 元素，因此这里就会把文本插入到 `<p>` 和 `</p>` 之间。（如果标签间原来有内容，原来的内容会被覆盖。）

- `;`
  非常重要的分号，表示一行代码结束。连缀完成。

5.1.3 平稳交接

很多 D3 方法都返回选中的元素（实际上是选中的元素的引用），正因为这样才能实

现方法连缀。一般来说，方法返回的都是正在操作的元素的引用，但有时候也不是。

所以要记住一点：在连缀方法时，次序很重要。每个方法的输出必须与下一个方法期待的输入匹配。如果相邻的输出和输入不匹配，那么结果就像在接力赛跑中把接力棒扔到地上一样。

要知道哪个方法期待什么输入，就要多参考 API 文档（https://github.com/mbostock/d3/wiki/API-Reference）。文档中有对每个方法的详细说明，包括是否返回选中的元素。

5.1.4 不要连缀

不用连缀语法也可以实现同样的功能：

```
var body = d3.select("body");
var p = body.append("p");
p.text("New paragraph!");
```

哎哟！太乱了。不过，有时候还不得不断开连缀，因为一口气串起来很多方法可能并不实际。或者，你只是希望让代码更容易组织，更符合自己的想法。

知道了怎么通过 D3 在页面中创建新元素，下面就该学习为元素附加数据了。

5.2 绑定数据

什么是绑定，为什么要绑定数据？

数据可视化说到底就是把数据映射到图形，数据入而图形出。也许是数值越大条形越长，也许特殊类别会显示为更亮的颜色。总之，映射规则你说了算。

在 D3 中，为了实现映射规则，需要把数据输入的值绑定到 DOM 中的元素上。绑定的意思类似于把数据“附加”或关联到特定的元素，以便将来引用数据的值和应用映射规则。如果没有绑定这一步，那么 DOM 元素就是一堆没用的东西。

5.2.1 怎么绑定

要通过 D3 的 `selection.data()` 方法把数据绑定到 DOM 元素。但必须具备两个条件：

- 数据；
- 选中的 DOM 元素。

下面就分别讨论一下。

5.2.2 数据

D3 可以很智能地处理各种数据，能够接受任何数值、字符串或对象（对象中包含着其他数组或键 / 值对）的数组，能够流畅地处理 JSON（和 GeoJSON），甚至还有一个内置的方法用于加载 CSV 文件。

简单起见，现在我们只以一个包含 5 个数值的数组为例：

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

如果你胆子大，而且有一些 CSV 或 JSON 数据，也可以直接按照下面说的做。否则，可以看一看 5.3 节。

1. 加载CSV数据

CSV 的意思就是逗号分隔的值（Comma-Separated Value）。CSV 数据文件一般是这样的：

```
Food,Deliciousness
Apples,9
Green Beans,5
Egg Salad Sandwich,4
Cookies,10
Vegemite,0.2
Burrito,7
```

这个文件中每一行都有两个值，值与值用逗号隔开。第一行一般作为标头，充当每一“列”的列名。

如果你的数据保存在 Excel 文件里，那它多半也是以这种行、列结构保存的。要想将它转换成 D3 可以使用的格式，先用 Excel 打开它，然后选择“另存为……”，再选择 CSV 文件类型。

假设把前面的 CSV 数据保存在 food.csv 中，那么就可以用 D3 的 `d3.csv()` 方法来加载它：

```
d3.csv("food.csv", function(data) {
    console.log(data);
});
```

`csv()` 接受两个参数：表示 CSV 文件路径的字符串和用作回调函数的匿名函数。回调函数只有在把 CSV 文件加载到内存之后才会执行。因此可以确定，在调用这个函数时，`d3.csv()` 已经把数据加载完了。

回调函数在执行时，会接收到加载和解析后的 CSV 数据。姑且把它命名为 `data` 吧，你叫它什么都行。剩下的事情就都交给回调函数办了。在前面的例子中，回调

函数只是把接收到的数组数据输出到了控制台，以便验证，如图 5-3 所示。（参见 03_csv_loading_example.html。）

```
[▶Object , ▶Object , ▶Object , ▶Object , ▶Object , ▶Object ]
```

图 5-3：输出到控制台的数组数据

可以看到数据保存在数组中（因为两头儿各有一个方括号），而且有 3 个元素，每个元素都是一个对象。点开每个对象旁边的小三角，可以看到它们的值，如图 5-4 所示。

```
[▼Object                    , ▼Object                    , ▼Object                         ,
    Deliciousness: "9"          Deliciousness: "5"          Deliciousness: "4"
    Food: "Apples"              Food: "Green Beans"         Food: "Egg Salad Sandwich"
  ▶__proto__: Object          ▶__proto__: Object          ▶__proto__: Object
 ▼Object                    , ▼Object                    , ▼Object                    ]
    Deliciousness: "10"         Deliciousness: "0.2"        Deliciousness: "7"
    Food: "Cookies"             Food: "Vegemite"            Food: "Burrito"
  ▶__proto__: Object          ▶__proto__: Object          ▶__proto__: Object
```

图 5-4：扩展开的数组元素

哇！每个对象都有 `Food` 和 `Deliciousness` 属性，与 CSV 文件中的值是一一对应的。（还有第三个属性，叫 `__proto__`，这个属性与 JavaScript 处理对象有关，现在不用管它。）D3 利用了 CSV 文件的第一行作为属性名，后续行作为属性值。这样一来就为我们节省了很多时间。

另外，还要注意：CSV 中的每个值都是以字符串形式保存的，连数字都是。（因为数字 9 两边有双引号，即是 `"9"`，而非 `9`。）这一点可能会影响后面的操作，因为你会把它当数值，但实际上它却是一个字符串。

处理数据加载错误

`d3.csv()` 是一个异步方法。什么意思？就是说在它加载数据的同时，其他 JavaScript代码会照样执行。（D3中其他加载外部资源的方法也都一样，比如 `d3.json()`。）

异步行为有时候不太好理解，因为我们可能会主观地认为CSV文件已经加载完了，但事实上还没有。此时常见的错误，就是在回调函数外面引用外部数据。记住，为了免得日后头疼，只能在回调函数内部（或回调函数调用的其他函数中）引用数据。

我个人喜欢先声明一个全局变量，然后再调用 `d3.csv()` 加载数据。在回调函数中，把数据复制给这个全局变量（以方便后面的函数使用该数据），最后再调用依赖数据显示图形的其他函数。例如：

```
var dataset;  // 全局变量

d3.csv("food.csv", function(data) {
    dataset = data;     // 加载完毕，就将其复制到 dataset.
    generateVis();      // 再调用其他依赖数据
    hideLoadingMsg();   // 显示图形的函数
});
```

还有一点更不好理解，回调函数无论是否成功加载数据文件，都会执行。网络连接可能断开，文件名可能拼错，或者Web服务器出了什么毛病，都可能导致数据加载失败。但无论如何，回调函数都会执行。如果加载数据失败，那再调用依赖数据的函数，可能就会在控制台中看到错误，图表也不会出现。这种情况很少发生，但知道如何处理很重要。

可以在回调函数的定义中增加一个可选的error参数。如果加载文件遇到问题，error中将包含Web服务器返回的错误消息，data就是undefined。如果文件加载成功，没发生错误，那么error的值就是null，data将保存着相应的数据。注意，error作为参数必须放在第一位，data是第二个：

```
var dataset;

d3.csv("food.csv", function(error, data) {

      if (error) {  // 如果 error 不是 null，肯定出错了
          console.log(error);  // 输出错误消息
      } else {       // 如果没出错，说明加载文件成功了
          console.log(data);   // 输出数据

          // 包含成功加载数据后要执行的代码
          dataset = data;
          generateVis();
          hideLoadingMsg();
      }

});
```

在 d3.csv() 的回调函数中验证数据很方便，但要在数据加载完毕后继续构建可视化图表，那最好还是再调用其他函数，比如：

```
var dataset;  // 声明全局变量

d3.csv("food.csv", function(data) {

    // 把 csv 数据交给全局变量
    // 以方便将来使用这些数据
    dataset = data;

    // 调用生成可视化图表的
    // 其他函数，比如：
    generateVisualization();
```

```
        makeAwesomeCharts();
        makeEvenAwesomerCharts();
        thankAwardsCommittee();

});
```

再提醒一下：如果你的数据是 TSV 文件，可以试试 d3.tsv() 方法。这个方法的其他方面与 d3.csv() 都一样。

2. 加载JSON数据

关于 JSON 数据，后面我们还会仔细讲解。现在，只要知道使用 d3.json() 方法可以像使用 d3.csv() 一样加载 JSON 数据就行了：

```
d3.json("waterfallVelocities.json", function(json) {
    console.log(json);  //输出到控制台
});
```

这里，我把解析后的输入命名为 json，你可以随便命名。

5.2.3 作出你的选择

数据讲完了，下面就以这个简单数组为例：

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

接下来需要决定选择什么。换句话说，你想让哪个元素与数据关联？同样，为简单起见，我们假设每个段落显示一个值。那么，你可能觉得代码应该这样来写：

```
d3.select("body").selectAll("p")
```

你或许没有错，但别忘了，我们想要选择的段落还都不存在呢。这就提出了使用 D3 时的一个常见的问题：怎么选择还不存在的元素？别急，因为答案可能会让你费点脑筋。

答案就在 enter() 这个真正的魔术方法里。先看下面的代码，稍后解释：

```
d3.select("body").selectAll("p")
    .data(dataset)
    .enter()
    .append("p")
    .text("New paragraph!");
```

打开示例代码中的 04_creating_paragraphs.html，可以看到 5 个新段落，每个段落中的内容都一样，如图 5-5 所示。

```
New paragraph!
New paragraph!
New paragraph!
New paragraph!
New paragraph!
```

图 5-5：动态生成的段落

下面是详细的解释。

- `d3.select("body")`

选择 DOM 中的 body 元素，把它交给连缀方法中的下一个方法。

- `.selectAll("p")`

  选择 DOM 中的所有段落。因为还没有段落，所以返回空元素。可以认为这个空元素代表马上就会创建的段落。

- `.data(dataset)`

  解析并输出数据值。dataset 数组中有 5 个值，因而此后的所有方法都将执行五遍，每次针对一个值。

- `.enter()`

  要创建新的绑定数据的元素，必须使用 enter()。这个方法会分析当前选择的 DOM 元素和传给它的数据，如果数据值比对应的 DOM 元素多，就创建一个新的占位元素。然后把这个新占位元素的引用交给链中的下一个方法。

- `.append("p")`

  取得由 enter() 创建的空占位元素，并把一个 p 元素追加到相应的 DOM 中。太棒了！然后它再把自己刚创建的元素交给链中的下一个方法。

- `.text("New paragraph!")`

  取得新创建的 p 元素，插入文本值。

5.2.4 绑定及确定

好的！现在已经读取、解析了数据，并将数据绑定到了在 DOM 中创建的 p 元素。不信？打开 04_creating_paragraphs.html，然后看看 Web 检查器，如图 5-6 所示。

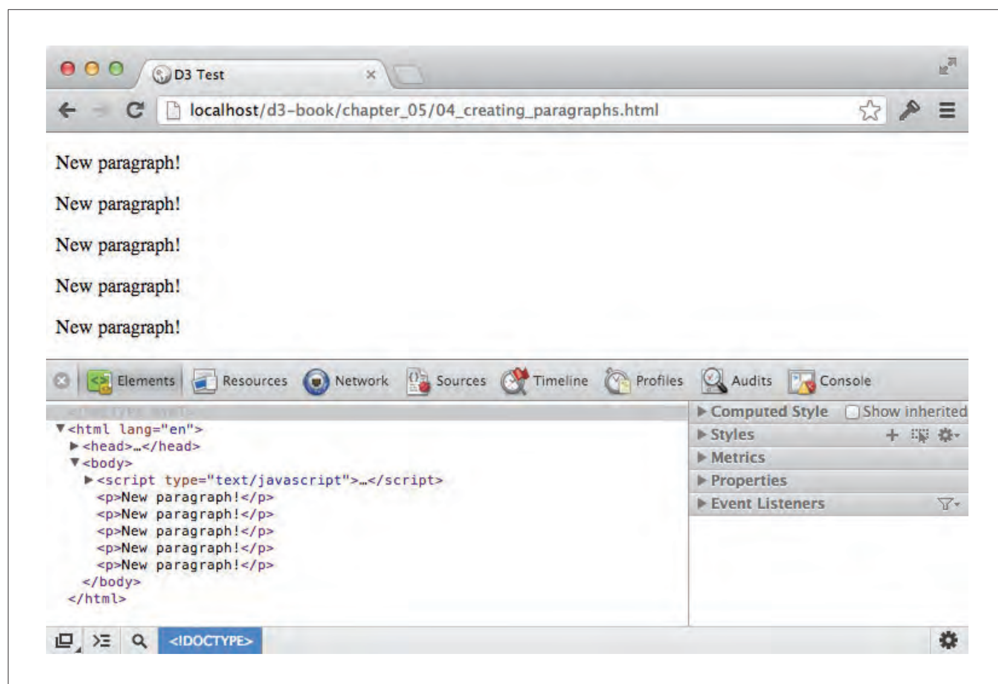


图 5-6：Web 检查器显示了新创建的元素

好，确实有 5 个段落，但数据在哪儿呢？切换到 JavaScript 控制台，输入以下代码，按回车。结果如图 5-7 所示。

```
console.log(d3.selectAll("p"))
```

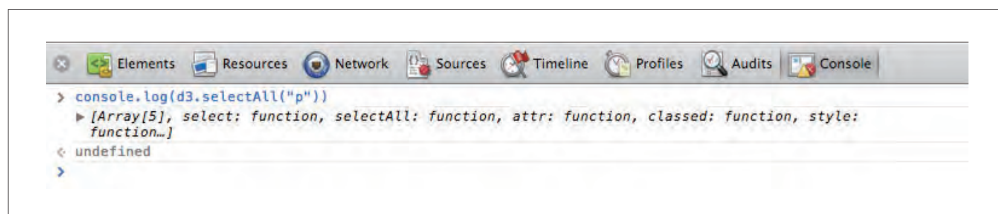


图 5-7：控制台中的输出

有一个数组！确切地说，是包含另一个数组的数组。单击灰色三角查看数组内容，如图 5-8 所示。

可以看到 5 个 p，编号从 0 到 4。单击第一个（编号为 0）旁边的三角，可以看到如图 5-9 所示的结果。

看到了吗，你看到了吗？太激动了，我都快忍不住了。在这儿呢！（见图 5-10）。

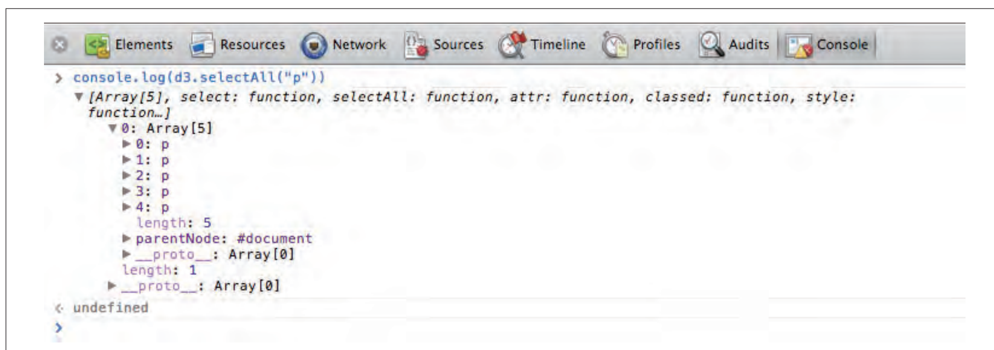


图 5-8：展开后的数组的数组

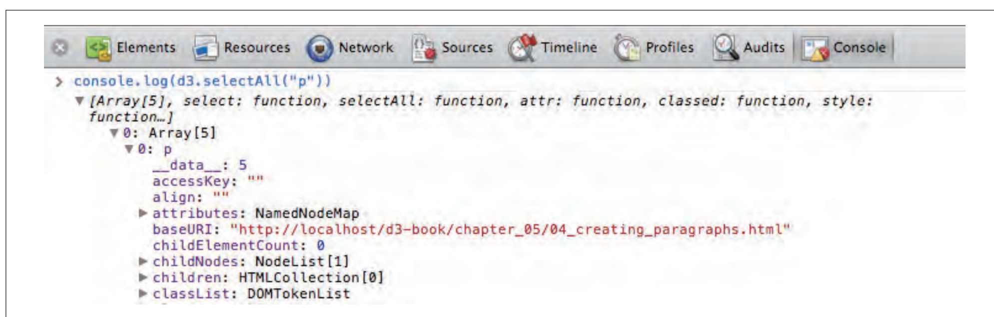


图 5-9：扩展后的 p 元素

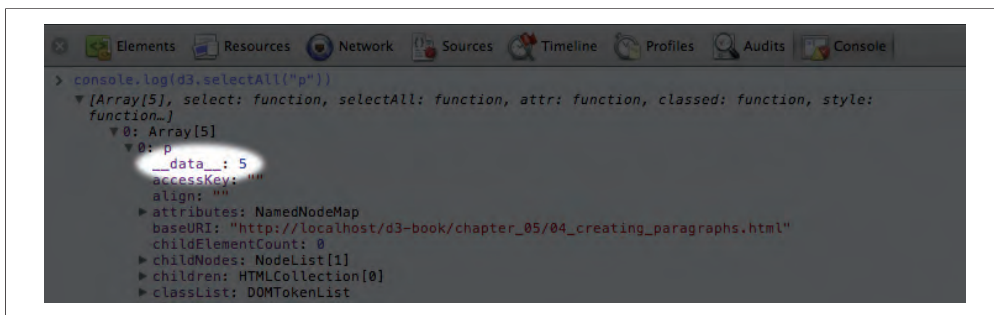


图 5-10：最后一步，绑定的数据

数据中的第一个值，数字 5，出现在了第一个段落的 `__data__` 属性中。单击其他段落，也可以看到它们的 `__data__` 属性分别包含 10、15、20 和 25。

你看，D3 绑定的数据没有出现在 DOM 中，而是作为该元素的 `__data__` 属性保存于内存中。控制台就是确认数据绑定是否正确的一个工具。

数据准备好了，下面该考虑对它做点什么了。

5.3　使用自己的数据

我们看到数据已经加载到页面中，而且也绑定到了 DOM 中新创建的元素。但怎么利用这些数据呢？以下是目前为止的代码：

```
var dataset = [ 5, 10, 15, 20, 25 ];

d3.select("body").selectAll("p")
    .data(dataset)
    .enter()
    .append("p")
    .text("New paragraph!");
```

把最后一行改一改：

```
    .text(function(d) { return d; });
```

看看包含新代码的 05_creating_paragraphs_text.html，结果应该如图 5-11 所示。

```
5

10

15

20

25
```

图 5-11：显示数据的段落

哇喔！我们用数据填充了每个段落，机关都在 `data()` 方法里。在连缀方法中，只要调用 `data()` 了，就可以随时创建一个接收 `d` 为输入的匿名函数。与当前元素对应，方法 `data()` 确保了每个 `d` 都会被赋予原始数据集中的一个值。

随着 D3 遍历每个元素，“当前元素”的这个值也会跟着变化。比如，循环到第三次时，代码会创建第三个 `p` 元素，而 `d` 就会被赋予数据集中的第三个值（即 `dataset[2]`）。于是，第三个段落的文本就是“15”。

5.3.1　自定义函数

一般我们在写新函数（也叫方法）的时候，基本的结构如下所示：

```
function(input_value) {
    // 完成一些计算
    return output_value;
}
```

前面我们使用的函数极其简单，没什么可说的：

```
function(d) {
    return d;
}
```

这种函数就叫*匿名函数*（anonymous function），因为它没有名字。相对而言，把函数保存在一个变量中，那个函数就是*命名函数*（named function）：

```
var doSomething = function() {
    //执行某些操作的代码
};
```

使用 D3 的过程中会写大量匿名函数。匿名函数是访问个别数据值并计算动态属性的关键所在。

我们例子中的匿名函数写在了 D3 的 `text()` 函数中，因此它会先执行，然后把执行结果交给 `text()`。而 `text()` 完成最后的工作（将接收到的参数值作为文本插入到选中的 DOM 元素中）：

```
.text(function(d) {
    return d;
});
```

不过，我们可以多做一点，因为这个函数是我们自己可以控制的，可以在里面写任何代码。没错，能写自己的代码既是好事，也是坏事。比如，可以像下面这样加上几个字：

```
.text(function(d) {
    return "I can count up to " + d;
});
```

这样就得到了如图 5-12 所示的结果，参见示例文件 06_creating_paragraphs_counting.html。

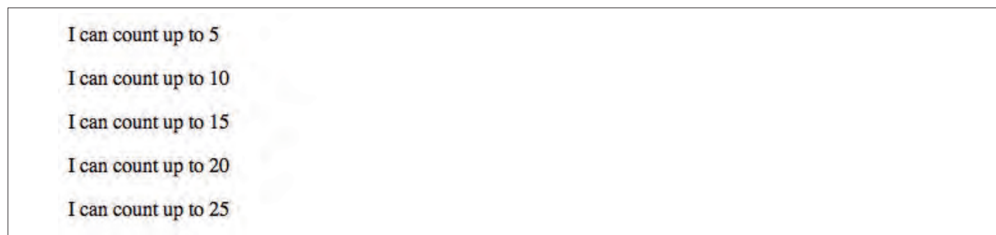


图 5-12：修改后的段落文本

5.3.2　数据需要拥抱

有读者可能会问：为什么要写成 `function(d) { ... }`，而不是直接传入 `d` 呢？比如，这样：

```
.text("I can count up to " + d);
```

此时此刻，如果不把 d 封装在匿名函数里，d 就没有值。可以想象成这个寂寞的小占位符需要一点温暖，包括来自和蔼可亲的函数圆括号的拥抱。（千万不要想太多，否则你会觉得匿名函数的拥抱简直要吓死人了，而且还会把问题复杂化。）

因此，需要把数据值轻轻地放到一个函数的怀抱里：

```
.text(function(d) {  // <-- 注意，温柔的拥抱在左边
    return "I can count up to " + d;
});
```

实际原因是 D3 中 `text()`、`attr()`，还有很多其他方法，都可以接收函数作为参数。比如，`text()` 既可以接收一个静态的字符串作为参数

```
.text("someString")
```

也可以接收某个函数的返回值：

```
.text(someFunction())  // 一般来说，someFunction() 应该也会返回字符串
```

或者一个匿名函数写成这样也可以作为参数：

```
.text(function(d) {
    return d;
})
```

最后传入的是匿名函数。如果 D3 发现它是一个函数，就会调用它，同时将当前数据值 d 作为参数传进去。这里把参数命名为 d 只是为了方便，你也可以叫它 `datum` 或 `info` 或其他任何名字。D3 只关心要有一个参数，有一个参数才能把当前数据值传进来。本书将全部使用 d，因为它很简洁，而且与 D3 的很多在线示例一致。

任何情况下，没有那个函数，D3 将无法把当前数据值传出来。没有匿名函数及其接收数据值的参数 d，D3 会不知所措，甚至会号啕大哭。（D3 比你想象得更容易激动。）

一开始的时候，你可能会觉得为了拿到 d 的值这么做有点笨，因为多费了一些事。但随着处理的数据越来越复杂，你会慢慢发现这种方法的重要性。

5.2.3 添加样式

了解 D3 的其他方法越多，就会发现越有意思。比如，`attr()` 和 `style()` 可以分别用来设置取得的元素的 HTML 属性和 CSS 属性。

比如，在前面代码中再加一行：

```
.style("color","red");
```

就会得到图 5-13 所示的结果，可以参考示例文件 07_creating_paragraphs_with_style.html。

I can count up to 5

I can count up to 10

I can count up to 15

I can count up to 20

I can count up to 25

图 5-13：段落文本变成红色了

所有文本都变成了红色，太好了。不过，我们可以再写一个自定义函数，只让那些数值大于某个阈值的文本显示为红色。于是，可以把代码中的最后一个字符串替换成一个函数，改成下面这样：

```
.style("color", function(d) {
    if (d > 15) {    //阈值是15
        return "red";
    } else {
        return "black";
    }
});
```

打开示例文件 08_creating_paragraphs_with_style_functions.html，就可以看到图 5-14 所示的结果。

I can count up to 5

I can count up to 10

I can count up to 15

I can count up to 20

I can count up to 25

图 5-14：动态为段落文本添加样式

好，前三行文本是黑色的，而数值超过 15 的后两行文本都变成了红色的。

这一章到现在，我们已经学习了加载数据、动态创建绑定到该数据的 DOM 元素。我想，接下来该学习绘制数据了！

第6章

基于数据绘图

这一章学习基于数据绘制图形。

现在，仍然以这个最简单的数据集为例：

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

6.1 绘制DIV

我们要使用这几个值来生成最简单的条形图。条形图实际上就是矩形，而 HTML 的 `div` 元素是绘制矩形的最简单手段。（同样，对浏览器来说，**一切都是矩形**。所以，你也可以把这个例子中的 `div` 换成 `span` 或其他别的元素试试看。）

严格来说，**柱形**图（column chart）指的是矩形沿垂直方向度量的图形，而沿水平方向度量的矩形叫**条形**图（bar chart）。不过，大多数人都不区分这两个概念，而统一称其为条形图，我们也沿用这一惯例。如图 6-1 所示，`div` 作为表示数据的条形是很合适的。



图 6-1：其貌不扬的 **div**

```
<div style="display: inline-block;
```

```
                width: 20px;
                height: 75px;
                background-color: teal;"></div>
```

对于讲 Web 标准的人来说，这样做可就犯了大忌。一般来说，不能为了展示而使用空 div。不过，对我们的例子而言，这属于例外。

因为是一个 div，所以它的宽度和高度都通过 CSS 样式设定。对生成图表而言，除了高度，其他属性都应该是共享的。为此，我们把公共的属性放到一个叫 bar 的类里面，放在文档头部的 style 标签中：

```
div.bar {
    display: inline-block;
    width: 20px;
    height: 75px;   /* 后面会覆盖这个高度 */
    background-color: teal;
}
```

接下来应该给每个 div 都应用 bar 类，以便这些 CSS 规则产生作用。如果通过手工来做，需要这么写：

```
<div class="bar"></div>
```

而使用 D3，给元素添加类要使用 selection.attr() 方法。理解 attr() 和相似的 style() 的区别很重要：attr() 设定 DOM 属性的值，而 style() 直接给元素添加 CSS 样式。

6.1.1 设定属性

attr() 用于设定 HTML 元素的属性和值。而 HTML 属性就是包含在尖括号 <> 中的任意属性 / 值对。比如，下面的标签

```
<p class="caption">
<select id="country">
<img src="logo.png" width="100px" alt="Logo" />
```

总共包含 5 个属性（以及对应的值），这些属性和值都可以通过 attr() 设定：

属　性	值
class	caption
id	country
src	logo.png
width	100px
alt	Logo

比如，要给某个元素添加一个 bar 类，可以这样写代码：

```
.attr("class", "bar")
```

6.1.2 关于类

元素的类作为 HTML 属性存在于标记代码中，同时 CSS 样式规则也可以引用它。由于为元素设定类（并据以推断样式）和直接给元素应用样式存在区别，有读者可能不知道使用哪种方式更好。D3 支持这两种为元素添加样式的方式。虽然用哪种方式取决你自己，但对于设定多个元素共享的样式，我还是建议使用设定类的方式，而对于一些特殊的样式，可以直接应用样式。（实际上，一会儿我们就会直接应用样式。）

这里还要简单介绍 D3 的另一个方法 classed()，用于快速地添加或删除元素的类。比如，前面那行代码可以重写成下面这样：

```
.classed("bar", true)
```

这行代码会给选中的元素添加类 bar。如果第二个参数是 false，则会从元素中删除类 bar：

```
.classed("bar", false)
```

6.1.3 言归正传

把数据集也考虑进来，则到现在为止完整的 D3 代码如下：

```
var dataset = [ 5, 10, 15, 20, 25 ];

d3.select("body").selectAll("div")
    .data(dataset)
    .enter()
    .append("div")
    .attr("class", "bar");
```

要知道现在的效果，可以在浏览器中打开 01_drawing_divs.html，查看一下源代码，再看看 Web 检查器。你会看到 5 个垂直的 div 条，分别对应数据集里的 5 个值。可是，条与条之间没有距离，看起来就像一个矩形，如图 6-2 和图 6-3 所示。



图 6-2：5 个 div 连成一片

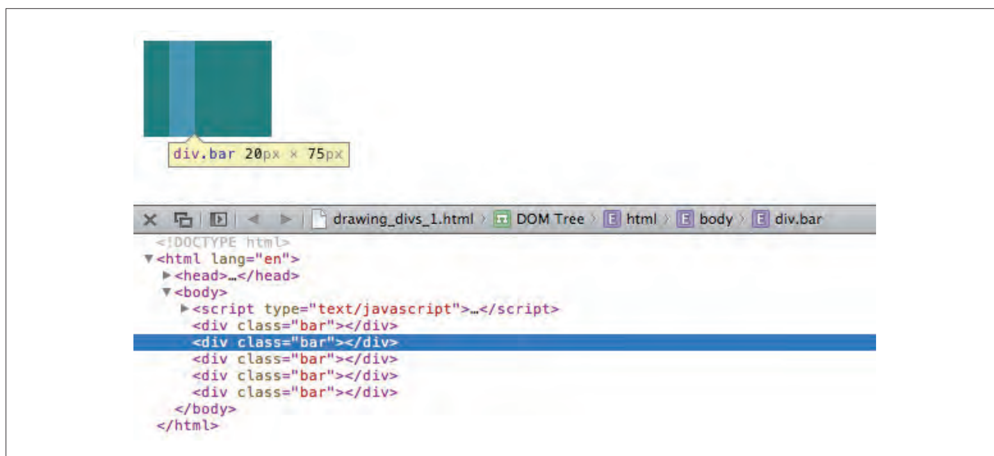


图 6-3：从 Web 检查器可以看出来，一共 5 个条

6.1.4　设定样式

前面说过，`style()` 方法用于直接为 HTML 元素应用 CSS 属性和值。这个方法执行的结果等价于在 HTML 的 `style` 属性中直接写入 CSS 规则：

```
<div style="height: 75px;"></div>
```

要生成条形图，每个条的高度必须是对应数据值的函数。下面我们就把这个映射关系添加到前面的 D3 代码中（注意每个方法连缀结束后都有一个分号）：

```
.style("height", function(d) {
    return d + "px";
});
```

在浏览器中打开 02_drawing_divs_height.html，可以看到条形都非常矮，如图 6-4 所示。

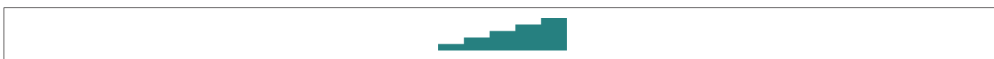


图 6-4：条形太短了

D3 在循环每个数据点的同时，`d` 会得到对应的值。因此，可以把 `d` 的（当前）值作为条形的高度值，后面再加个 px（指定单位为像素）。结果 5 个条形的高度分别是：5px、10px、15px、20px 和 25px。

条形图太短了没意思，还是让它们都长高点吧：

```
.style("height", function(d) {
    var barHeight = d * 5;  // 放大 5 倍
    return barHeight + "px";
});
```

再给条形之间增加一些距离（写在文档头部的嵌入式 CSS 样式），让它们彼此分开一点：

```
margin-right: 2px;
```

不错！拿着这张图可以直接参加 SIGGRAPH[1] 大会了（见图 6-5）。

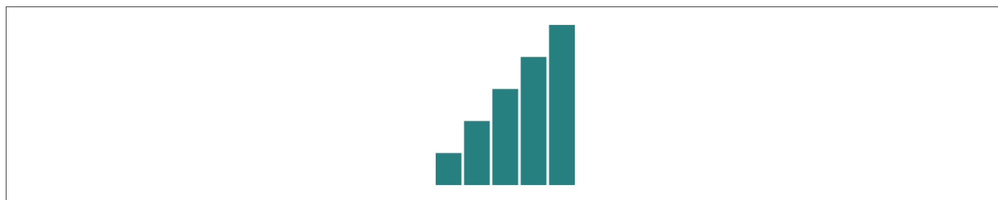


图 6-5：增高后的条形图

试一试 03_drawing_divs_spaced.html 吧。同样，看看源代码，再通过 Web 检查器对照一下 HTML 和最终的 DOM。

6.2 data()的魔力

这个条形图还不错，但真正的数据永远不会那么清爽：

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

下面我们再把示例数据弄得乱一些，参见 04_power_of_data.html：

```
var dataset = [ 25, 7, 5, 26, 11 ];
```

数据的变化也体现在了条形图上，如图 6-6 所示。当然，也不是只能有 5 个数据点啊。好吧，再多加一些！（参见 05_power_of_data_more_points.html。）

```
var dataset = [ 25, 7, 5, 26, 11, 8, 25, 14, 23, 19,
                14, 11, 22, 29, 11, 13, 12, 17, 18, 10,
                24, 18, 25, 9, 3 ];
```

注 1：SIGGRAPH，是美国计算机协会的计算机图形专业组（Special Interest Group on Computer Graphics）。——译者注

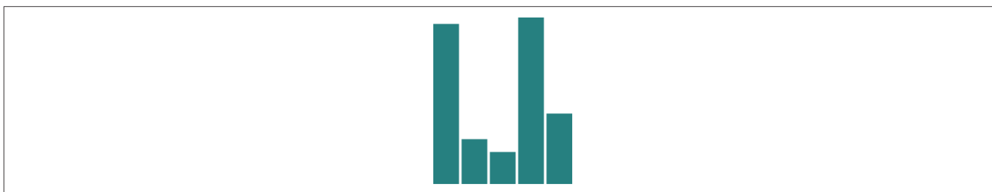


图 6-6：接近真实的数据值

25 个数据点了，不是 5 个了（图 6-7）！

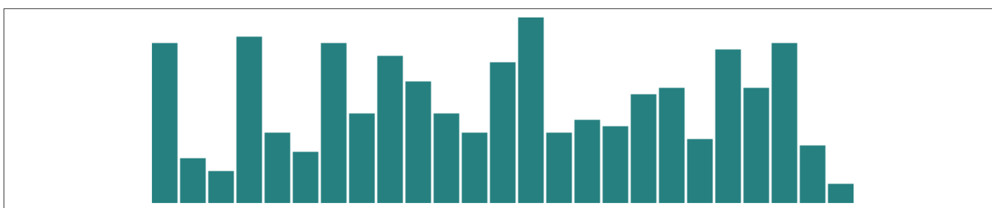


图 6-7：更多的数据值

D3 是怎么按照需要自动扩展图表的呢？

```
d3.select("body").selectAll("div")
    .data(dataset)  // <-- 答案在这里！
    .enter()
    .append("div")
    .attr("class", "bar")
    .style("height", function(d) {
        var barHeight = d * 5;
        return barHeight + "px";
    });
```

传给 data() 的是 10 个值，它就会循环 10 次。传给它 100 万个值，它就会循环 100 万次（不过你得有点耐心）。

这就是 data() 的魔力，它能遍历你扔给它的任何长度的数据集，然后依次执行连缀在后面的每一个方法，同时更新每个方法执行时的上下文，以确保 d 永远引用循环中当前的数据值。这句话可能太长了点，如果你没理解，过一会儿就能理解了。建议大家复制 05_power_of_data_more_points.html，随便修改一下数据值，然后看看条形图怎么变化。

记住，是*数据*驱动可视化，而不是相反。

随机数据

有时候，为了好玩，也可以生成随机数据值来进行验证。示例 06_power_of_data_

random.html 中就是这么做的，每次重新加载页面，条形图都会来个大变脸，如图 6-8 所示。

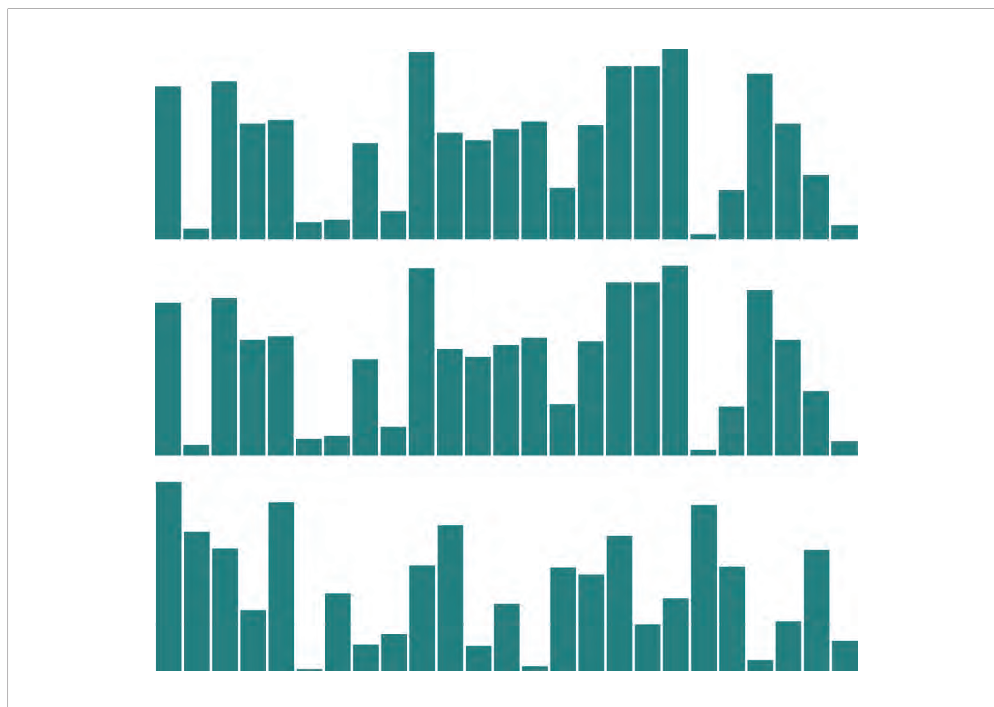


图 6-8：随机生成的条形图

查看源代码，下面这几行是关键：

```
var dataset = [];                              // 初始化空数组
for (var i = 0; i < 25; i++) {                 // 循环 25 次
    var newNumber = Math.random() * 30;        // 生成介于 0 到 30 之间的随机数
    dataset.push(newNumber);                   // 把新数值添加到数组中
}
```

这里没有使用 D3 的任何方法，只有 JavaScript。简单解释一下，以上代码做了以下几件事。

1. 创建了一个名为 `dataset` 的空数组；
2. 初始化一个 `for` 循环，执行 25 次；
3. 每次生成一个介于 0 到 30 之间的随机数（严格来说，最大的数要接近 30。`Math.random()` 返回的最小值是 0.0，最大值不会达到 1.0。如果返回值是 0.99999，那么 0.99999 乘以 30 就是 29.9997，或者比 30 只小那么一丁丁点儿的数）；

4. 把新数值追加到数组中（`push()` 是数组的方法，每次执行都会把一个新值推进数组末尾）。

为了更好玩一点，打开 JavaScript 控制台，输入 `console.log(dataset)`，应该可以看到随机生成的那 25 个数值，如图 6-9 所示。

```
> console.log(dataset)
  [14.793717765714973, 21.65710132336244, 22.01914135599509, 10.693866850342602,
  8.197558452375233, 8.327909619547427, 9.349913026671857, 6.715130957309157,
  20.352523955516517, 20.892786516342312, 18.432767554186285, 7.062793713994324,
  11.519823116250336, 8.91862049465999, 5.422192756086588, 8.956057007890195,
  13.239774140529335, 24.165618284605443, 14.453229457139969, 27.792113937903196,
  2.717762708198279, 12.752952876035124, 1.7288982309401035, 21.01240729680285,
  26.07524922117591]
```

图 6-9：控制台中显示的随机数值

注意，这些都是小数值或浮点值（比如 14.793717765714973），没有我们一开始指定的整数或整型数（如 14）。对我们例子来说，小数值完全没有问题。但假如你需要整数，可以使用 JavaScript 的 `Math.round()` 或 `Math.floor()` 方法。前者将数值向上舍入为最接近的整数，后者则总是向下舍入。比如，可以把生成随机数的代码：

```
var newNumber = Math.random() * 30;
```

放到 `Math.floor()` 方法的调用中：

```
var newNumber = Math.floor(Math.random() * 30);
```

以上代码可以保证生成的新数值一定是介于 0 到 29（包含）之间的整数。为什么没有 30？因为 `Math.random()` 永远返回小于 1.0 的值，而 `Math.floor()` 永远向下舍入，因此 29 就是可能返回的最大值。

在浏览器中打开 07_power_of_data_rounded.html，通过控制台验证一下所有数值确实都变成了整数，如图 6-10 所示。

```
> console.log(dataset)
  [23, 19, 18, 16, 24, 29, 1, 5, 13, 4, 29, 23, 11, 9, 16, 10, 15, 4, 28, 23, 13,
  19, 20, 20, 27]
```

图 6-10：控制台中显示的随机整数

用 `div` 展示数据的内容都讲完了。下一节我们看看 SVG 给我们提供了哪些可能性。

6.3 绘制SVG

要了解 SVG 的语法，请参考 3.8 节。

关于 SVG 元素，最关键是要记住它们的各个方面都是通过属性来设定的。换句话说，就是通过标签中的属性 / 值对来指定 SVG 元素的各方面特征，比如：

```
<element property="value"></element>
```

嗯，跟 HTML 一样！

```
<p class="eureka">Eureka!</p>
```

前面已经体验过 D3 的 `append()` 和 `attr()` 方法了，它们分别用于创建新 HTML 元素和设定它们的属性。因为 SVG 元素存在于 DOM 中，跟其他 HTML 元素一样，因此生成 SVG 图形仍然要使用 `append()` 和 `attr()` 方法。

6.3.1 创建SVG

首先要创建一个 SVG 元素，以便在其中保存所有图形：

```
d3.select("body").append("svg");
```

这行代码先找到文档的 `body` 元素，然后在结束的 `</body>` 标签前添加一个新的 `svg` 元素。不过，我建议稍微改一改这行代码：

```
var svg = d3.select("body").append("svg");
```

还记得 D3 的大多数方法都会返回它们所操作的 DOM 元素的引用吗？上面这行代码就把 `append()` 返回的新元素保存在了变量 `svg` 中。有了这个引用，将来就可以少写很多代码，因为不用总是写 `d3.select("svg")`，而只要写 `svg` 即可：

```
svg.attr("width", 500)
   .attr("height", 50);
```

当然，也可以把所有代码都写在一行：

```
var svg = d3.select("body")
              .append("svg")
              .attr("width", 500)
              .attr("height", 50);
```

请大家参见 08_drawing_svgs.html 中代码。打开检查器，看看 DOM 中是不是已经有了新 SVG 元素了？没错，有一个空的 SVG 元素。

为了方便后续的编码，建议把宽度和高度值也保存在变量中，参考 09_drawing_svgs_size.html。在源代码里可以看到如下代码：

```
//宽度和高度
```

```
var w = 500;
var h = 50;
```

在后面的所有例子里，我都会这么做的。因为把尺寸值保存在变量里，将来就可以方便地引用，如下：

```
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)   // <-- 看这里
            .attr("height", h); // <-- 还有这里!
```

同样，如果你想把某个字符串也保存在变量里，我一点也没意见。

6.3.2　数据驱动的图形

现在该绘制一些图形了。还是拿我们忠实的老数据集作例子吧：

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

然后，使用 `data()` 迭代每个数据点，为它们分别创建一个圆形：

```
svg.selectAll("circle")
    .data(dataset)
    .enter()
    .append("circle");
```

记住，`selectAll()` 会返回对所有圆形的空引用（因为还不存在呢），而 `data()` 把数据绑定到即将创建的元素，`enter()` 返回对这个新元素的占位引用，最后 `append()` 把圆形添加到 DOM。在这里，代码会在 SVG 元素的末尾依次追加所有圆形，因为一开始我们选择的是 `svg`（而不是原来的 `body`）。

为方便以后引用圆形，可以创建一个新变量来保存它们的引用：

```
var circles = svg.selectAll("circle")
                 .data(dataset)
                 .enter()
                 .append("circle");
```

非常好，但这些圆形还需要位置和大小信息，如图 6-11 所示。小心点，下面这几行代码可能会把你看晕：

```
circles.attr("cx", function(d, i) {
            return (i * 50) + 25;
        })
       .attr("cy", h/2)
       .attr("r", function(d) {
            return d;
       });
```

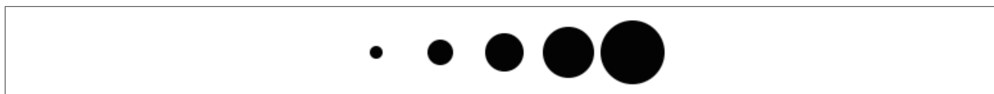


图 6-11：代表数据的圆形

还是参考一下 10_drawing_svgs_circles.html 吧。下面我们一步一步地讲解：

```
circles.attr("cx", function(d, i) {
            return (i * 50) + 25;
        })
```

这是通过引用所有圆形的变量来设置每一个圆形的 cx 属性。（在 SVG 中，cx 是圆形圆心的 *x* 坐标，没忘吧？）因为数据已经绑定到了圆形，所以对每个圆形来说，d 分别对应于原始数据集中相应的值（5、10、15、20 和 25）。

另一个值 i 也是 D3 替我们自动生成的。（谢谢 D3！）跟 d 一样，变量 i 也是随便想的一个名字，你也可以改，比如改成 counter 或 elementID。我喜欢 i，因为简单，而且它还跟 for 循环中的计数器变量 i 的命名一致，几乎所有在线示例中都使用 i 作计数器。

好啦，i 就是当前元素的索引值。这个值从 0 开始，因此第一个圆形的 i == 0，第二个圆形的 i == 1，依此类推。这里我们利用了这个顺序值，把每个圆形都向右推进一段，因为每次循环 i 的值都会加 1：

```
(0 * 50) + 25   //返回 25
(1 * 50) + 25   //返回 75
(2 * 50) + 25   //返回 125
(3 * 50) + 25   //返回 175
(4 * 50) + 25   //返回 225
```

为了在自定义函数里使用这个索引，必须记住把它作为函数的参数，比如 function(d, i)。当然，同时也要传入 d，即使你在当前函数里不用它也要传（就像前面例子中一样）。同样，这个参数的名字并不重要，但函数参数的个数（一个或两个）不能少。

什么，听到参数你就头大？不要紧的。实际上，你基本上只要记住 d 和 i 就足够了。后面我们也不会再学习其他匿名函数的参数了。好，下一行：

```
.attr("cy", h/2)
```

cy 是每个圆形圆心的 *y* 轴坐标。这里是把 cy 设置成了 h 的一半。还记得吧，h 保存着整个 SVG 元素的高度，因此 h/2 等于把所有圆形垂直居中。

```
    .attr("r", function(d) {
        return d;
    });
```

最后，每个圆形的半径 r 被设置为等于 d，反映数据值的大小。

6.3.3　你好，色彩

色彩填充（fill）和描边（stroke）同样也是属性，也可以通过 attr() 方法来设定。比如，再连缀如下代码：

```
    .attr("fill", "yellow")
    .attr("stroke", "orange")
    .attr("stroke-width", function(d) {
        return d/2;
    });
```

就可以得到图 6-12 所示的彩色圆形了，参考一下 11_drawing_svgs_color.html 吧。

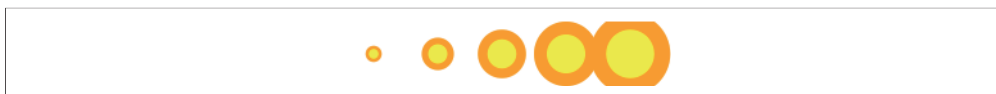


图 6-12：彩色的数据圆形

当然，通过混合各种属性和自定义函数可以得到各种效果。不过，数据可视化的关键在于选择适当的**映射规则**，从而保证反映数据的视觉元素能够让用户容易看懂，对用户有价值。

6.4　绘制条形图

接下来，我们要把前面所学的东西整合起来，用 SVG 来创建简单的条形图。

为此，就从修改 div 条形图的代码开始，用 SVG 代替之。SVG 为视觉表现提供了更多的灵活性。之后，我们还要学习给图形添加标签，以便看到确切的数据值。

6.4.1　老方法生成的条形图

现在看看 12_making_a_bar_chart_divs.html，其中有用老方法 div 生成条形图的代码：

```
var dataset = [ 5, 10, 13, 19, 21, 25, 22, 18, 15, 13,
                11, 12, 15, 20, 18, 17, 16, 18, 23, 25 ];

d3.select("body").selectAll("div")
    .data(dataset)
    .enter()
```

```
        .append("div")
        .attr("class", "bar")
        .style("height", function(d) {
            var barHeight = d * 5;
            return barHeight + "px";
        });
```

图 6-13 展示了在浏览器中看到的效果。或许你很难想象，我们完全可以大幅改进这个条形图。

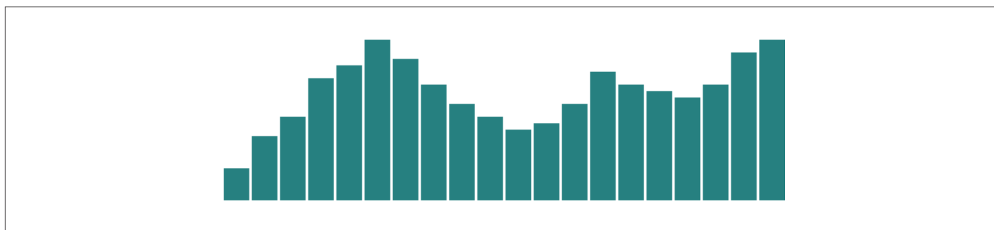


图 6-13：基于 div 的条形图

6.4.2 用新方法改进条形图

首先，需要确定新 SVG 元素的大小：

```
// 宽度和高度
var w = 500;
var h = 100;
```

当然，你可以把 w 和 h 改成其他名字，比如 svgWidth 和 svgHeight。你觉得怎么明确，就怎么命名。JavaScript 程序员非常注重效率，因此你会经常看到一些单个字母的变量，代码间也没有空格，很难看懂却能很快写出来。

然后，告诉 D3 创建空 SVG 元素，并将其添加到 DOM 中：

```
// 创建 SVG 元素
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);
```

就算是啰嗦一点吧，这些代码会在结束的 </body> 的标签前面插入新的 svg 元素，将其宽度和高度设置为 500 像素和 100 像素。同时，代码还把返回的结果保存在了变量 svg 中，因此后面可以方便地引用这个 SVG 元素，而不必每次再使用 d3.select("svg") 之类的代码重新选择。

接下来，不创建 div，而生成矩形元素 rect 并将它们添加到 svg 中：

```
svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   .attr("x", 0)
   .attr("y", 0)
   .attr("width", 20)
   .attr("height", 100);
```

这行代码选择了 svg 中的所有矩形。当然，这时候什么都还没有呢，所以会返回一个空的元素集。（很奇怪呀，没错，不过相信我。在 D3 中，永远得先选择你想要操作的元素，即使这个元素或元素集暂时还没有。）

接下来 data(dataset) 看到了数据集中有 20 个值，就把这些值交给了 enter() 处理。然后，enter() 会为每个数据值返回一个占位元素，让它们都有对应的尚未创建的 rect。

根据这 20 个占位元素，append("rect") 会分别把它们插入 DOM 中。第 3 章也介绍过，每个 rect 必须有 x、y、width 和 height 属性。这里就是用 attr() 为每个新创建的 rect 设置了这些属性。

漂亮吗，不？好吧，或许不漂亮。所有条形都已经生成了（在 Web 检查器中看看 13_making_a_bar_chart_rects.html 的 DOM 中有什么），但它们的坐标和大小全都一样，所以就重叠在了一起，如图 6-14 所示。当然，这时候的条形并没有反映数据。

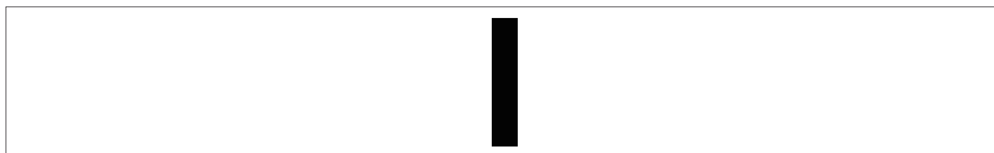


图 6-14：孤零零的一个条形

首先来解决重叠的问题。为此，要把 x 值从 0 改为一个与 i（也就是每个值在数据集中的位置序号）对应的动态生成的值。让第一个条形的 *x* 轴坐标是 0，随后的分别是 21、42，依此类推。（第 7 章将介绍 D3 的比例尺，那是一个解决这个问题的更灵活的机制。）

```
.attr("x", function(d, i) {
    return i * 21;  // 条形宽 20 像素，外加 1 像素间距
})
```

完整的代码请参考 14_making_a_bar_chart_offset.html，结果如图 6-15 所示。

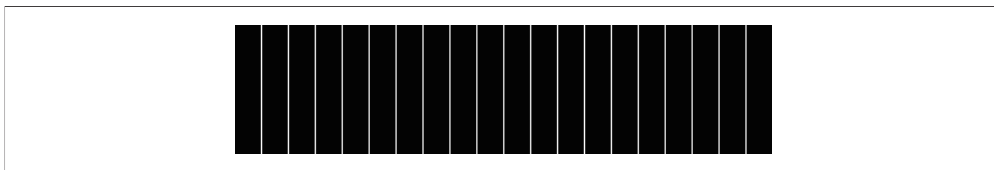


图 6-15：20 个条形

起作用啦，但这样做不是很灵活。如果数据集再大一些，那么条形会更多，而最右边的条形很可能跑到 SVG 外头去！算一算吧，每个条形宽 20 像素，外加 1 像素间距，那么 500 像素宽的 SVG 只能容纳 23 个条形。图 6-16 演示了第 24 个条形跑到 SVG 外面去的情状。

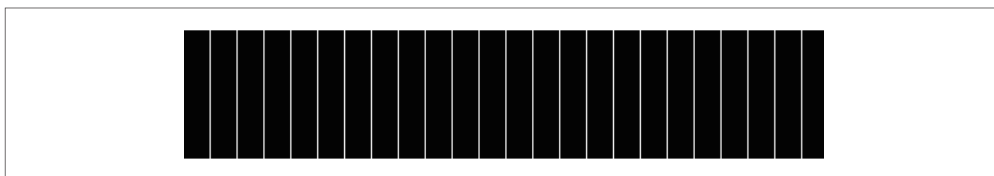


图 6-16：24 个条形

更好的做法是使用灵活、动态的坐标，让所有可见图形的高度、宽度、*x* 坐标、*y* 坐标，全部能根据数据成比例地缩放。

跟任何编程任务一样，实现动态缩放的办法有无数种。但我们只能选择一种。首先，从改进设置每个条形 *x* 坐标的那行代码开始修改：

```
.attr("x", function(d, i) {
    return i * (w / dataset.length);
})
```

好啦，现在所有条形的 *x* 坐标值都直接与 SVG 的宽度（w），以及数据集中数据值的个数（dataset.length）紧紧关联在一起了。这样一来，所有条形就会在 SVG 中均匀分布，无论数据集中是有 20 个值（如图 6-17 所示）



图 6-17：20 个平均分布的条形

还是只有 5 个值，如图 6-18 所示。

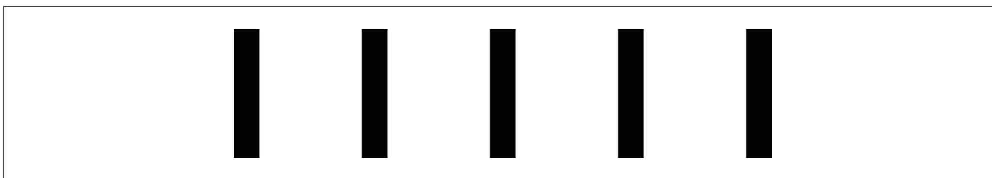


图 6-18：5 个平均分布的条形

15_making_a_bar_chart_even.html 中包含到目前为止所有的代码。

现在，该设置条形的宽度，让宽度也成比例缩放。如果数据多，条形就窄一些，如果数据少，条形就宽一些。为此，要在声明 SVG 宽度和高度的地方再声明一个新变量：

```
//宽度和高度
var w = 500;
var h = 100;
var barPadding = 1;  // <-- 新变量!
```

然后在设置每个条形宽度的那一行代码里引用这个变量。换句话说，每个条形宽度不再是固定的 20 像素，而是要设置成 SVG 宽度与数据值个数的商再减掉间距值：

```
.attr("width", w / dataset.length - barPadding)
```

起作用啦！（参见图 6-19 和 16_making_a_bar_chart_widths.html。）

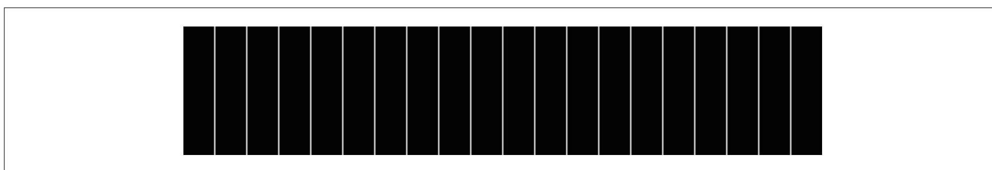


图 6-19：20 个平均分布的条形，而且宽度可以动态调整

条形的宽度和 *x* 轴坐标都能正确调整，无论是像上面那样有 20 个数据值，还是有 5 个数据值（参见图 6-20），甚至有 100 个数据值（参见图 6-21）。

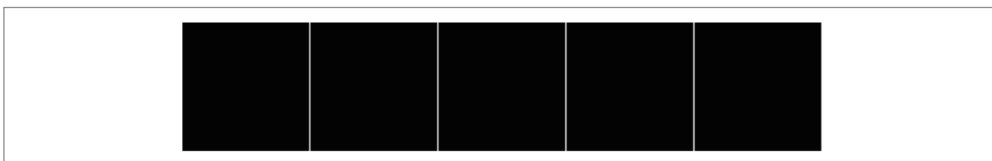


图 6-20：5 个平均分布的条形，而且宽度可以动态调整

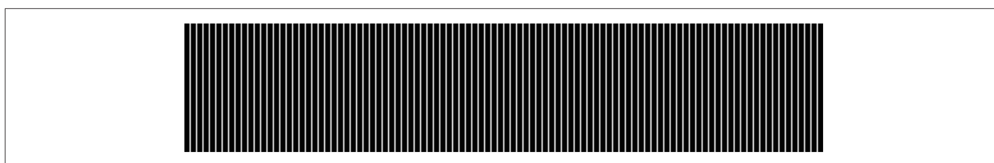


图 6-21：100 个动态宽度和间距的条形

最后，还要通过代码让数据值来决定条形高度。聪明的读者可能会想，只要在设置条形高度时使用 d 的值就好了：

```
.attr("height", function(d) {
    return d;
});
```

哎哟，图 6-22 中的条形图看起来有点其貌不扬啊。要不把数值放大几倍？

```
.attr("height", function(d) {
    return d * 4;  // <-- 放大 4 倍!
});
```

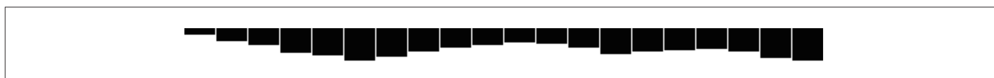


图 6-22：动态高度

唉，看来没那么简单！我们想要的不是像图 6-23 所示的上端对齐，而是要下端对齐——不过别埋怨 D3，这是 SVG 的问题。



图 6-23：放大高度

或许读者还记得，在绘制 SVG 矩形时，x 和 y 值指定的是它们左上角的坐标。换句话说，每个条形的原点或者参照点都是它的左上角。所以呢，如果能把坐标指定为每个条形图的左下角就简单多了。可是，SVG 只支持左上角坐标系，而且坦白来讲，SVG 在这个问题上跟我们的想法不一样。

既然条形只能“从上向下长”，那么相对于 SVG 的上沿，每个条形的“上沿”在哪里呢？好，每个条形的上沿可以用 SVG 高度与对应的数据值之间的关系来表示，就像这样：

```
.attr("y", function(d) {
    return h - d;  // 高度减数据值
})
```

然后，为了让每个条形的“下沿”与 SVG 的下沿对齐（参见图 6-24），每个 `rect` 的 `height` 可以就设置为数据值本身：

```
.attr("height", function(d) {
    return d;  // 原数据值
});
```

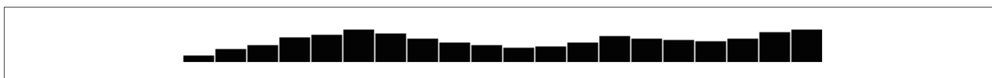


图 6-24：条形变成从下向上“长”了

现在可以把 `d` 改成 `d * 4`，把条形放大一些，结果就变成了如图 6-25 所示。（使用 D3 的比例尺机制，可以更适当地排布条形，不过现在还没到介绍它的时候。）

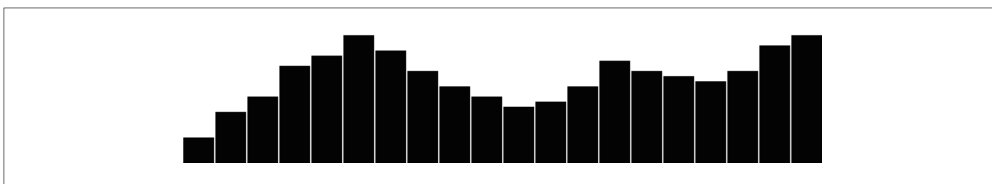


图 6-25：条形变长了

这个自下而上的 SVG 条形图的代码位于 17_making_a_bar_chart_heights.html 中。

6.4.3 上色

添加颜色很容易，只要用 `attr()` 方法设置 `fill` 属性就行了：

```
.attr("fill", "teal");
```

结果如图 6-26 所示，所有条形都是青色（`teal`），代码在 18_making_a_bar_chart_teal.html 中。

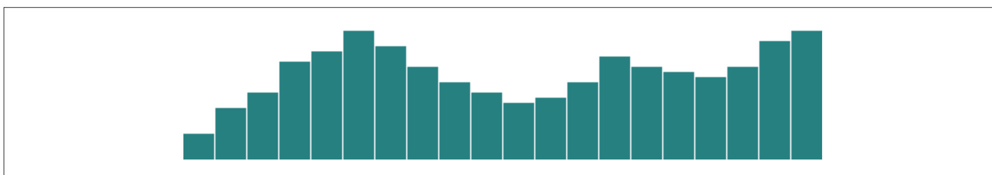


图 6-26：青色的条形图

青色虽然不难看，但最好能让颜色反映数据的某些特性。换句话说，最好可以根据数据值来编码颜色值。（对于这个条形图而言，这样做叫做**双重编码**，即同样的数据值被编码成两种可见的特性：条形高度和颜色。）

通过数据生成颜色也很简单，同样只要写一个接收 d 作为参数的自定义函数即可。这里，我们把 "teal" 替换成一个自定义函数，结果就是 6-27 所示的条形图：

```
.attr("fill", function(d) {
    return "rgb(0, 0, " + (d * 10) + ")";
});
```

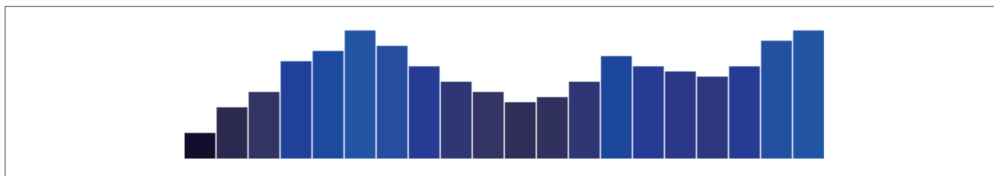


图 6-27：数据驱动的蓝色条形图

相关代码可以参见 19_making_a_bar_chart_blues.html。这种视觉编码好像用处不大，但从这个例子能知道怎么把数据转换成颜色。这里是把 d 乘以 10，然后将结果作为 rgb() 函数的最后一个参数（蓝色分量）。因此，值越大（条形越长），蓝色分量越多；值越小（条形越短），蓝色分量越少（接近黑色）。此时的红、绿分量的值都是 0。

多值映射

注意一下，你会发现在方法链上已经调用了5次attr()，分别设定了条形的x、y、width、height和fill值。

如果你觉得这样一遍一遍地输入attr()很麻烦，那肯定会喜欢D3的多值映射机制。这个机制让你一次性设置多个值，而且仍然是使用attr()方法。假设要把一个圆形平移到SVG左上角，再设置成红色，可以每次单独调用attr()：

```
svg.select("circle")
   .attr("cx", 0)
   .attr("cy", 0)
   .attr("fill", "red");
```

或者，也可以把这三个属性和它们的值都封装在一个对象中，然后统一交给attr()：

```
svg.select("circle")
```

```
    .attr({
        cx: 0,
        cy: 0,
        fill: "red"
    });
```

后一种写法更加简洁，如果你用过jQuery，可能早就熟悉这种把属性/值对打包为对象的语法了。每个属性/值对中的值也可以是匿名函数，而且可以像往常一样通过引用d和i动态生成值。利用这种多值映射机制，可以把我们前面条形图的代码重写如下：

```
svg.selectAll("rect")
    .data(dataset)
    .enter()
    .append("rect")
    .attr({
        x: function(d, i) { return i * (w / dataset.length); },
        y: function(d) { return h - (d * 4); },
        width: w / dataset.length - barPadding,
        height: function(d) { return d * 4; },
        fill: function(d) { return "rgb(0, 0, " + (d * 10) + ")";}
    });
```

如果你喜欢这种写法，那么听说style()和其他一些方法也支持多值映射，你一会更开心。不过本书还是会使用比较冗长的写法，只要你知道还有其他选择就好。

6.4.4 加标签

条形图很好，但如果再配上实际的数据值，那就更好了。这时候就得用到本节要介绍的标签了，而且通过 D3 生成标签是非常非常容易的。或许你还有印象，在 SVG 中是可以添加文本元素的。好，现在开始吧：

```
svg.selectAll("text")
    .data(dataset)
    .enter()
    .append("text")
```

看着眼熟？对，怎么添加矩形，就怎么添加文本。首先，选择想要的元素，然后绑定数据，加入元素（此时的元素只是占位符），最后把新文本元素添加到 DOM 中。

接下来使用 text() 方法让每个文本元素都包含一个数据值：

```
    .text(function(d) {
        return d;
    })
```

再进一步扩展，通过设置 x 和 y 值来定位文本元素。偷个懒，把前面用于条形的代码复制过来也可以：

```
.attr("x", function(d, i) {
     return i * (w / dataset.length);
})
.attr("y", function(d) {
     return h - (d * 4);
});
```

好啦！数据值标签来了。但有些数字显示不完整，参见图 6-28。

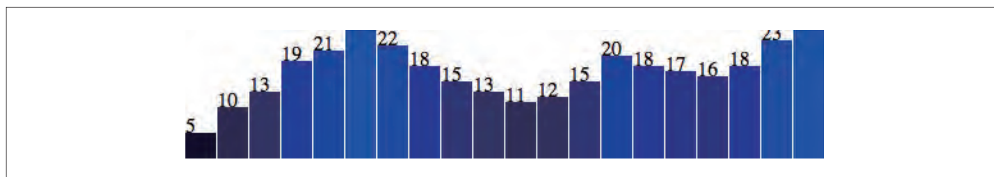


图 6-28：值标签新鲜出炉

对，需要把值标签向下移动一小段距离，需要进行一点坐标计算：

```
.attr("x", function(d, i) {
     return i * (w / dataset.length) + 5;  // +5
})
.attr("y", function(d) {
     return h - (d * 4) + 15;              // +15
});
```

得到的图 6-29 所示的结果好多了，但看不清楚了。

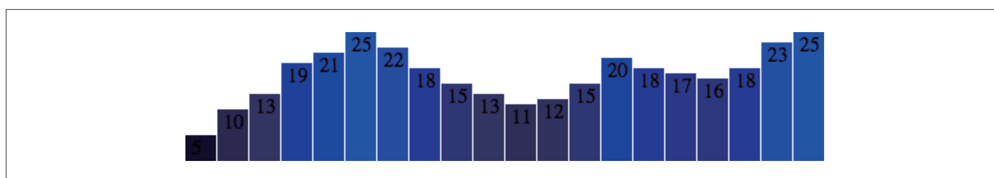


图 6-29：条形内的值标签

好在这个问题容易解决：

```
.attr("font-family", "sans-serif")
.attr("font-size", "11px")
.attr("fill", "white");
```

代码真神奇啊！图 6-30 中的图形可以在 20_making_a_bar_chart_labels.html 中看到。

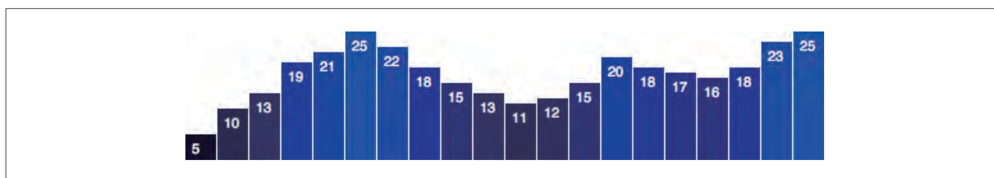


图 6-30：漂亮的值标签

假如你没有排版强迫症，现在就可以收手了。不过，假如你跟我一样吹毛求疵，会发现值标签与各自的条形没有完美对齐。（比如，第一个条形中的 5。）改这个太容易啦，可以使用 SVG 的 text-anchor 属性来水平居中文本，就像指定 x 值一样：

```
.attr("text-anchor", "middle")
```

然后，再改一下计算 x 值的方法，让它等于每个条形的左边位置值加上条形宽度的一半：

```
.attr("x", function(d, i) {
    return i * (w / dataset.length) + (w / dataset.length - barPadding) / 2;
})
```

另外，为了让间距也合适，我还要把标签向上提一个像素，结果如图 6-31 所示，完整的代码请参考 21_making_a_bar_chart_aligned.html：

```
.attr("y", function(d) {
    return h - (d * 4) + 14;  //15 变成了 14
})
```

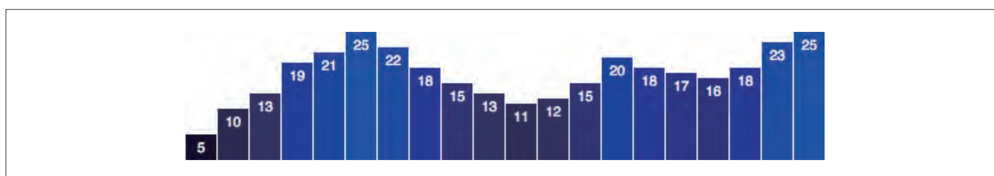


图 6-31：居中值标签

6.5 绘制散点图

迄今为止，我们只学习了绘制条形图，而且使用的是简单的一维数据值。如果是两组数据，需要互相对照着绘制，那就需要用到第二维。散点图是在两个坐标轴上表现两组对应值的常见图表。

6.5.1 数据

第 3 章曾介绍过，组织数据的方式有很多种，非常灵活。对于散点图，我们可以使

用数组的数组。主数组包含每个数据“点”的元素，而每个“点”元素同样也是数组，但只包含两个值：*x* 坐标和 *y* 坐标。

```
var dataset = [
                [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
              ];
```

别忘了，`[]` 表示数组，因此嵌套的方括号 `[[]]` 表示数组中的数组。数组的元素以逗号分隔，因此包含三个数组的数组，形式上应该是这样的：`[[],[],[]]`。

如果给上面的数据集多加点空白，看起来会更清楚：

```
var dataset = [
                  [ 5,     20 ],
                  [ 480,   90 ],
                  [ 250,   50 ],
                  [ 100,   33 ],
                  [ 330,   95 ],
                  [ 410,   12 ],
                  [ 475,   44 ],
                  [ 25,    67 ],
                  [ 85,    21 ],
                  [ 220,   88 ]
              ];
```

这样就很明白了，每行数据都对应图形中的一个点。比如，`[5, 20]`，其中 5 是 *x* 轴坐标，20 是 *y* 轴坐标。

6.5.2 散点图

事实上，可以借鉴绘制条形图时所用的大部分代码，包括创建 SVG 元素的部分：

```
// 创建 SVG 元素
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);
```

但这里不是要创建 `rect` 元素，而是要为每个数据点创建 `circle`：

```
svg.selectAll("circle")  // <-- 不是 "rect" 了
   .data(dataset)
   .enter()
   .append("circle")     // <-- 不是 "rect" 了
```

当然，也不用再设定矩形的 `x` 和 `y` 以及 `width` 和 `height` 属性，而是要设定圆形的 `cx`、`cy` 和 `r` 属性：

```
        .attr("cx", function(d) {
             return d[0];
        })
        .attr("cy", function(d) {
             return d[1];
        })
        .attr("r", 5);
```

图 6-32 中的散点图是由 22_scatterplot.html 生成的。

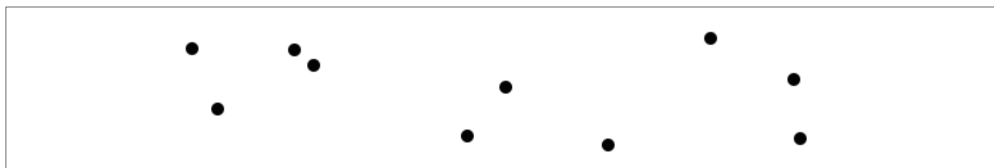


图 6-32：简单的散点图

注意这一次设定 cx 和 cy 属性时访问数据值的方式。此时，function(d) 中的 d 将保存着 D3 传过来的当前数据值，而且是大数组中的小数组。

每个数据值 d 本身也是一个数组（而不是 3.14159 这样的数值），就需要通过方括号访问它们的值。也就是说，不能再用 return d，而要用 return d[0] 和 return d[1]，它们分别返回小数组中的第一个和第二个元素。

比如，对于第一个数据点 [5, 20]，第一个值（位置为 0 的值）是 5，第二个值（位置为 1 的值）是 20：

```
d[0] // 返回 5
d[1] // 返回 20。
```

同样地，如果要访问大数组中的值（比如在 D3 代码外部），也要使用方括号，比如：

```
dataset[5] // 返回 [410, 12]
```

甚至可以使用多个方括号访问嵌套数组的值：

```
dataset[5][3] // 返回 12
```

不信？那就试一试 22_scatterplot.html，打开 JavaScript 控制台，在其中输入 dataset[5] 和 dataset[5][5]，看看返回值是什么。

6.5.3　散点大小

显然，你一定会想到让圆形的大小有所不同，让圆形的面积对应各自表现的值。一

般来说，在通过圆形表现数量时，通用的做法都是将数据点编码为面积，而不是半径。凭直觉想象一下，这就像用所有“墨水”或像素来反映数据值。将数据值映射为圆形半径是一个常见的错误。（我自己就犯过好多次这种错误。）映射到半径更简单，数学计算最少，但结果却会导致数据表现不准确。

可在创建 SVG 圆形时无法指定面积值。那就只能自己根据数据值先计算出对应的半径，然后再设置 `r` 属性。

知道了面积，怎么求半径？还记得圆形的面积公式吧，圆形面积等于 π 乘以半径的平方，或者 $A = \pi r^2$。

假设现在圆形的面积是 `d[1]`，而为了让上方的圆形更大一些，得用 `h` 减去它。因而现在的面积是 `h - d[1]`。（第 7 章将介绍怎么运用比例尺机制来更直观地实现这一点。）

为了把面积转换成半径，关键在于求平方根。JavaScript 恰好有一个内置的 `Math.sqrt()` 函数，比如：`Math.sqrt(h - d[1])`。

好了，下面重新设置所有 `r` 值：

```
.attr("r", function(d) {
    return Math.sqrt(h - d[1]);
});
```

图 6-33 显示了此时的结果，代码可以参见 23_scatterplot_sqrt.html。

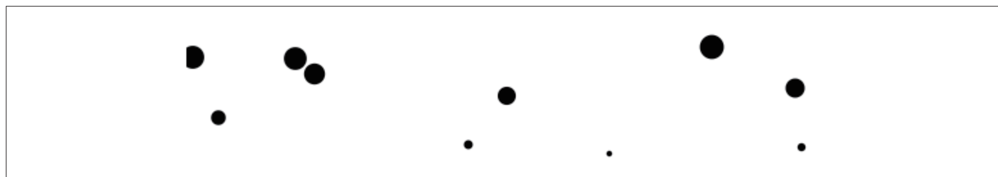


图 6-33：大小不同的散点图

这里随便用 SVG 高度 `h` 减去数据点的 *y* 坐标值 `d[1]` 之后，再求其平方根，结果是圆形 *y* 坐标越大（在图中位置越靠下），面积越小（半径越短）。

这种使用圆形面积来可视化数据的办法并不是特别好。毕竟，我们只是为了演示，即怎么使用 `d` 和使用方括号来引用个别的数据点，对数值应用一些变换，然后再将重新计算的值（这里就是 `r` 的值）返回给属性设定方法。

6.5.4　标签

接下来看一下怎么用 `text` 元素为数据点加标签。在此，同样可以采用之前绘制条

形图时的代码，从下面的代码开始：

```
svg.selectAll("text")  // <-- 注意是 "text"，而非 "circle" 或 "rect"
   .data(dataset)
   .enter()
   .append("text")     // <-- 这里也一样!
```

这行代码首先查找 SVG 中的所有 text 元素（还没有呢），然后把这些元素追加到每个数据点上。最后，text() 方法设定了每个元素的内容：

```
   .text(function(d) {
        return d[0] + "," + d[1];
   })
```

看起来有点乱？请容我解释。同样，这里仍然使用 function(d) 来访问每个数据点。然后，在匿名函数内部，使用 d[0] 和 d[1] 取得数据点数组的两个值。

这里的加号（+）对于字符串（比如引号中的逗号 ","）而言，是一个拼接操作符。因此这一行代码的作用就是把 d[0] 和 d[1] 的值拼起来，中间塞进去一个逗号。结果就像 5,20 或 25,67 这样。

接下来通过设定 x 和 y 值来定位文本。这里，我们暂且使用设定圆形圆心位置的 d[0] 和 d[1]：

```
   .attr("x", function(d) {
        return d[0];
   })
   .attr("y", function(d) {
        return d[1];
   })
```

最后，再给文本设定一些样式：

```
   .attr("font-family", "sans-serif")
   .attr("font-size", "11px")
   .attr("fill", "red");
```

结果如图 6-34 所示。虽然不算漂亮，但标签是添加成功了！代码请参见 24_scatterplot_labels.html。



图 6-34：带标签的散点图

6.6 更上一层楼

但愿经过本章的学习，大家对 D3 的核心概念已经有了初步的理解：加载数据、生成新元素，然后用数据值派生出这些元素的属性值。

没错，图 6-34 很难说是及格的数据可视化图形。那张散点图看起来有点费劲，而代码也没有灵活地运用数据。坦率地说，我们还没有超过——噢，天哪——Excel 的图表向导呢！

别急呀，D3 可比图表向导酷得多（更别提曲别针小助手了）。不过，要生成一个令人侧目的交互式图表，还需要我们继续深造，学习更多 D3 的知识。想灵活地运用数据？下一章我们就学习 D3 的比例尺。想让散点图一目了然？再下一章我们就学习 D3 的数轴生成器和数轴标签。

现在大家最好美美地休息一会儿，伸伸胳膊，弯弯腰。最好出去散散步，或者来杯咖啡，外加一个三明治。我会在这里等着你（如果你不介意的话），等你一回来，咱们就开始探索 D3 的比例尺！

第7章 比例尺

“比例尺是一组把输入域映射为输出范围的函数。”

这是 Mike Bostock 对 D3 中比例尺（scale）给出的定义，实际上要用一两句话说明白什么是比例尺并不容易。

一般来说，任意数据集中的值不可能恰好与图表中的像素尺度一一对应。比例尺就是把这些数据值映射为可视化图形中使用的新值的便捷手段。

D3 的比例尺就是那些你定义的带有参数的函数。定义好之后，就可以调用这些比例尺函数，传入值，它们就能返回按比例生成的输出值。你可以自己定义任意多个比例尺函数。

听到比例尺这个词，有人可能会不由自主地想到最终图表中的一系列刻度线，对应一系列值。不要搞错，这些刻度线是坐标轴的一部分，而坐标轴只是比例尺的一种形象的表示。比例尺实际上代表着一种数学关系，不可能直接输出可见的图形。大家可以把比例尺和坐标轴想象成两样不同但相关的东西。

本章只讨论**线性比例尺**，这是一种最常用也最容易理解的比例尺。理解了线性比例尺，其他什么序数、对数、平方根比例尺的，全是小菜一碟。

7.1 苹果和像素

假设以下数据集反映的是路边水果摊每月卖出的苹果数量：

```
var dataset = [ 100, 200, 300, 400, 500 ];
```

首先，这是个重大新闻，因为这个小小的水果摊每月居然都能多卖出 100 个苹果！生意果然兴隆。为了展示销售业绩，你想通过条形图来表现苹果销量直线上涨的趋势，即每个数据值对应着相应条形的高度。

到目前我止，我们一直都在直接显示数据值，忽略了单位的差异。换句话说，卖出 500 个苹果，相应的条形就是 500 像素高。

这样当然可以，但要是下个月卖出了 600 个苹果呢？要是一年后卖出 1800 个苹果呢？那用户可能就得换一台大显示器，才能看到完整的条形图！

这时候就要用到比例尺了，因为苹果不是用像素来衡量的（当然橙子也不行），所以就需要通过比例尺来表达和实现两者之间的映射。

7.2　值域和范围

比例尺的输入值域（input domain）指可能的输入值的范围。以前面苹果的数据为例，输入值域大致为 100 到 500（即数据集中最小和最大值），或者 0 到 500。

比例尺的输出范围（output range）指输出值可能的范围，一般以用于显示的像素为单位。输出范围完全取决于你，因为你是信息可视化的设计师。如果你想让最短的苹果条高为 10 像素，让最高的苹果条高为 350 像素，那就可以把输出范围设定为 10 到 350。

下面我们就来分析一下输入值域为 [100,500]，输出范围为 [10,350] 的情况。要把输入的最小值 100 映射到这个比例尺上，应该返回最小的值 10。如果输入为 500，它应该返回 350。要是输入为 300 呢？那输出就是 180。（因为 300 是值域的中值，而 180 是范围的中值。）

如图 7-1 所示，可以在平行的数轴中把值域和范围的对应关系绘制出来。

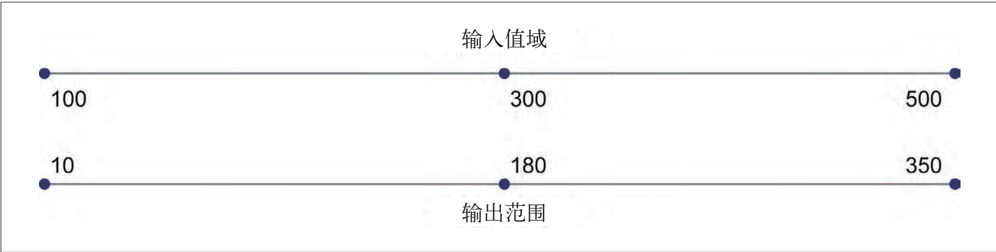


图 7-1：输入值域和输出范围的数轴

提醒一下，为了不让自己的大脑搞混输入值域和输出范围，建议你做个小游戏。我

说“输入”，你就说“值域”。我再说“输出”，你就说“范围”。准备好了吗？开始：

输入！值域！

输出！范围！

输入！值域！

输出！范围！

记住了？好。

7.3　归一化

如果你熟悉归一化（normalization）的概念，那太好了，因为对于线性比例尺来说，就是那个意思。

归一化就是根据可能的最小值和最大值，把某个数值映射为介于 0 和 1 之间的一个新值的过程。比如，一年 365 天，那么 310 天映射过来就是 0.85，即 85%。

对于线性比例尺，D3 可以帮我们处理归一化过程中的数学计算：输入值根据值域先进行归一化，然后再把归一化之后的值对应到输出范围。

7.4　创建比例尺

D3 有一个比例尺函数生成器，通过 `d3.scale` 来访问。要生成一个比例尺，在 `d3.scale` 后面加上要创建的比例尺类型即可。建议读者现在打开示例页面 01_scale_test.html，在控制台里试试下面的代码：

```
var scale = d3.scale.linear();
```

恭喜！现在 `scale` 就是一个可以接收参数的函数了。（不要被 `var` 迷惑了，虽然它在 JavaScript 中用于创建变量，但变量也可以保存函数。）

```
scale(2.5);  //Returns 2.5
```

因为还没有设定值域和范围，所以这个函数会按照 1:1 的比例映射输入和输出。换句话说，输入什么，输出仍然是什么。

设置比例尺的值域需要调用 `domain()` 方法，并将值域以数组形式传给它。假设值域是 100 到 500，那么就可以这样写代码：

```
scale.domain([100, 500]);
```

设定输出范围的方式类似，但要调用 range() 方法：

```
scale.range([10, 350]);
```

这些步骤可以独立完成，也可以连缀起来完成：

```
var scale = d3.scale.linear()
                    .domain([100, 500])
                    .range([10, 350]);
```

不管怎么设定，现在的比例尺已经准备就绪了！

```
scale(100);  //返回 10
scale(300);  //返回 180
scale(500);  //返回 350
```

一般来说，我们都会在 attr() 或其他类似方法中调用比例尺函数，而不会像这样独立调用它。好了，下面来看看怎么修改上一章散点图的代码，从而动态地应用比例尺。

7.5 缩放散点图

回顾一下创建散点图的数据集：

```
var dataset = [
                [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
              ];
```

还记得吧，这个数据集是用数组的数组表示的。我们是把每个数组的第一个值映射到了 *x* 轴，把第二个值映射到了 *y* 轴。现在要应用比例尺了，先从 *x* 轴开始。

目测一下 *x* 值，大致是从 5 到 480，因此合理的值域应该是 0 到 500，对吧？

为什么那么看着我？噢，你想让自己的代码更灵活更有伸缩性吧，这样即使将来数据集有变化也不用修改代码。你真聪明！你考虑得很周全，假如我们在给路边的那个苹果摊设计一个数据展示板，那么应该让代码适应苹果销量的大幅增长。而图表呢，无论是卖出 5 个苹果还是 100 成个苹果，也都应该能够正常显示。

7.5.1 d3.min()和d3.max()

既然不想给值域设置固定的值，那可以使用两个方便的数组函数：d3.min() 和 d3.max()，让它们帮你动态分析数据集。比如，下面的代码会循环数组中的每个 *x* 值，返回其中最大的那个：

```
d3.max(dataset, function(d) {
    return d[0];  // 引用嵌套数组中的第一个值
});
```

以上代码会返回 480，因为 480 是数据集中最大的 *x* 值。下面解释一下代码的执行过程。

D3 的 min() 和 max() 原理相同，都接受一到两个参数。第一个参数必须是对数组也就是数据集的引用。假如有一个简单的一维数组，如 [7, 8, 4, 5, 2]，那很明显就知道怎么比较这些值的大小，不需要第二个参数。比如：

```
var simpleDataset = [7, 8, 4, 5, 2];
d3.max(simpleDataset);  // 返回 8
```

max() 函数只是简单地循环数组中的每个值，然后找出其中最大的那个。

可是，我们的数据集并不是个简单的数值数组，而是一个数组的数组。调用 d3.max(dataset) 可能会得到不想要的结果：

```
var dataset = [
                  [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                  [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
              ];
d3.max(dataset);  // 返回 [85, 21]，什……么？
```

要告诉 max() 想比较哪个值，就必须传入第二个参数，也就是一个存取器函数：

```
d3.max(dataset, function(d) {
    return d[0];
});
```

这个存取器函数是一个匿名函数，max() 会把数组中的每个元素（即这里的 d）交给它。存取器函数的目的是指定比较哪个值。对我们的数据集而言，需要比较 *x* 值，也就是嵌套数组的第一个值，位置为 0。因此存取器函数就是这样的：

```
function(d) {
    return d[0];  // 返回嵌套数组中的第一个值
}
```

等等，这个函数的语法在生成散点图的时候见过啊，当时是用匿名函数来取得并返回值：

```
.attr("cx", function(d) {
    return d[0];
})
.attr("cy", function(d) {
    return d[1];
})
```

好记性，这是 D3 中一个常用的模式。用不了多长时间，你就会对代码中随处可见的匿名函数习以为常了。

7.5.2 设置动态缩放

把我们刚刚介绍的知识综合起来，就可以创建一个动态映射 *x* 轴值的比例尺函数：

```
var xScale = d3.scale.linear()
                     .domain([0, d3.max(dataset, function(d) { return
                         d[0]; })])
                     .range([0, w]);
```

首先，我们把这个比例尺函数命名为了 `xScale`。当然，你可以根据自己的需要来命名，但 `xScale` 可以帮我记住这个函数的作用。

其次，这里同时设定了值域和范围，使用的是包含两个值的数组。

第三，我们把值域的最小值设定为 0 了。（实际上，在这里也可以使用 `min()` 来生成动态的值。）而值域的最大值则设定为数据集中的最大值（目前是 480，将来可能会变）。

最后，注意一下输出范围被设定为 0 到 `w`（即 SVG 的宽度）。

接下来可以使用非常类似的代码为 *y* 轴创建比例尺函数：

```
var yScale = d3.scale.linear()
                     .domain([0, d3.max(dataset, function(d) { return d[1]; })])
                     .range([0, h]);
```

要注意的是，`max()` 方法这一次引用的是 `d[1]`，即嵌套数组的 *y* 值。类似地，`range()` 方法中的最大值设定成了 `h`，而不是 `w`。

好啦，比例尺函数定义完成！现在该把它们派上用场了。

7.5.3 整合缩放后的值

现在要做的就是在散点图代码的基础上，修改一下为每个值创建圆形的代码。比如，原来是

```
.attr("cx", function(d) {
    return d[0];  // 返回数据集中的原始值
})
```

为了使用缩放后的值（而不是原始值），要改成这样：

```
.attr("cx", function(d) {
```

```
        return xScale(d[0]);  // 返回缩放后的值
    })
```

同样，对于 *y* 轴的代码

```
    .attr("cy", function(d) {
        return d[1];
    })
```

要改成这样：

```
    .attr("cy", function(d) {
        return yScale(d[1]);
    })
```

为了比例协调，下面再对设置文本标签坐标的代码作相同修改，即原来的

```
    .attr("x", function(d) {
        return d[0];
    })
    .attr("y", function(d) {
        return d[1];
    })
```

要改成这样：

```
    .attr("x", function(d) {
        return xScale(d[0]);
    })
    .attr("y", function(d) {
        return yScale(d[1]);
    })
```

完事了！

完整的代码请参考 02_scaled_plot.html。视觉上呢，图 7-2 的结果可能跟最初的散点图一样令人失望！ 可是我们毕竟已经取得了进步，只不过表面上看不出来而已。

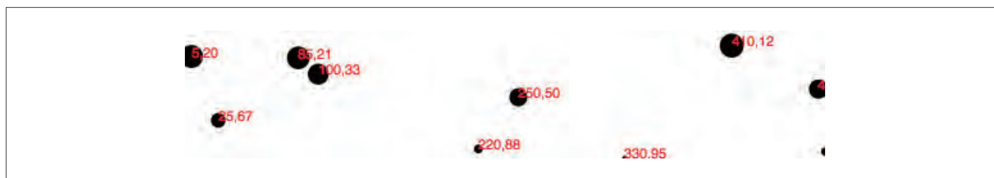


图 7-2：使用了 *x* 和 *y* 比例尺的散点图

7.6 修饰图表

注意到了吗，现在较小的 *y* 值在图表上方，而较大的 *y* 值在图表下方。既然我们使

用了 D3 的比例尺，那么要反转它们就易如反掌了。只要把 `yScale` 的输出范围由

```
.range([0, h]);
```

改为

```
.range([h, 0]);
```

即可。代码参见 03_scaled_plot_inverted.html，结果如图 7-3 所示。



图 7-3：反转 *y* 轴比例尺之后的散点图

没错，现在 `yScale` 对较小的输入会返回较大的输出值，这样才会把相应的圆形和文本推向下方，接近图形的底部。我知道，这没有什么难理解的！

可是有些圆形被切掉了一部分。为此，得引入一个边距变量：

```
var padding = 20;
```

以便在设置两个比例尺的时候加入边距。边距可以把圆形向里推，使它们远离 SVG 的四边，从而避免被切掉。

原来 `xScale` 的范围是 `range([0, w])`，现在改成这样：

```
.range([padding, w - padding]);
```

原来 `yScale` 的范围是 `range([h, 0])`，现在改成这样：

```
.range([h - padding, padding]);
```

这样就可以在 SVG 的上、下、左、右四边分别增加 20 像素的边距。的确如此，如图 7-4 所示！



图 7-4：加入了边距的散点图

但最右边的文本标签仍然会被切掉，那就把 `xScale` 的边距加倍吧，结果如图 7-5 所示：

```
.range([padding, w - padding * 2]);
```

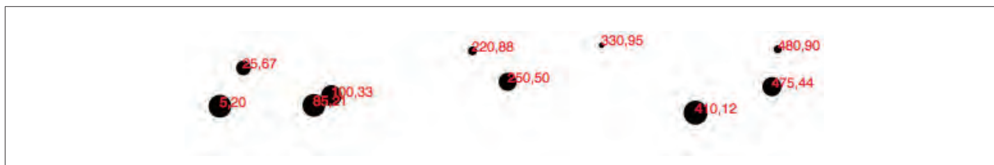


图 7-5：加入更大边距的散点图



这里讲的加入边距的方法很简单，但不够完善。因为我们需要能够更多地控制图表每个边（上、下、左、右）的边距。那么就有必要把指定这些值的方法提取出来，以便在不同项目中重用。虽然本书到现在一直没有采用 Mike Bostock 的外边距约定（http://bl.ocks.org/mbostock/3019563），我还是建议大家看一看，或许对你有用。

好多啦！完整代码可以参考 04_scaled_plot_padding.html。不过，我还想再改动一个地方。我们原来是把每个圆形的半径设置为 *y* 值的平方根（这多少有点走偏门的味道，因为对可视化帮助不大），为什么不为半径也创建一个自定义的比例尺呢？

```
var rScale = d3.scale.linear()
                     .domain([0, d3.max(dataset, function(d) { return
d[1]; })])
                     .range([2, 5]);
```

然后，再像这样设定圆形半径：

```
.attr("r", function(d) {
    return rScale(d[1]);
});
```

这太好了，因为可以保证所有圆形的半径永远都会介于 2 和 5 之间。（或许也有例外，可以参考后面介绍的 `clamp()`。）换句话说，数据值为 0（最小的输入）时，圆形半径为 2 像素（或直径为 4 像素）。而最大的数据值对应的圆形半径是 5 像素（或直径为 10 像素）。

看吧，图 7-6 展示了对视觉属性第一次应用比例尺的效果。（代码参见 05_scaled_plot_radii.html。）

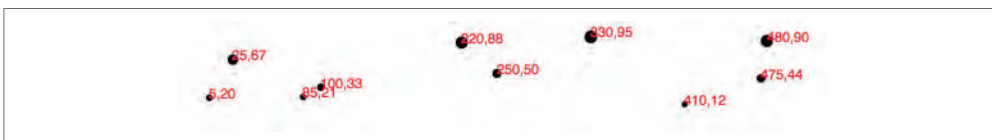


图 7-6：对圆形半径应用比例尺后的散点图

最后，我真担心比例尺的威力还没有让你开窍，所以想再给数据集增加一个数据点：`[600, 150]`。

怎么样？看看 06_scaled_plot_big.html 吧，效果如图 7-7 所示。注意啊，增加了右上角的数据点之后，原来的那些圆形都向左向下移动而且缩得更紧了，这样才能维护所有数据点之间的相对位置关系。

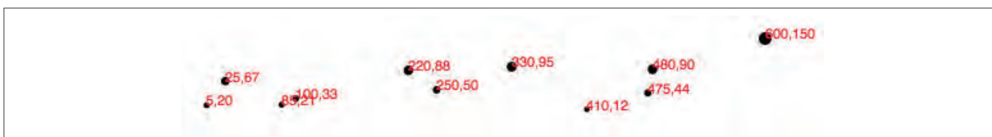


图 7-7：添加了大值之后的散点图

该向大家透露最后一个秘密了：现在，你可以随意修改 SVG 的大小，而图表中的一切都会随之成比例地缩放。在图 7-8 中，我们把 SVG 的高度值 `h` 从 100 增大到了 300，其他都没变。

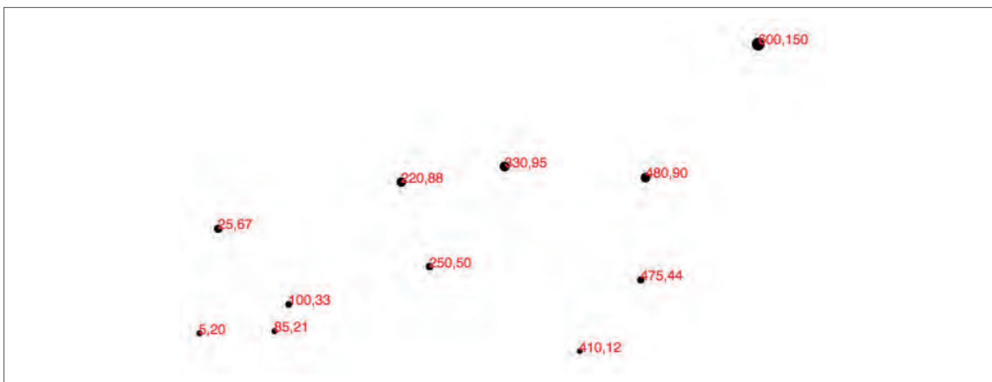


图 7-8：成比例放大后的散点图

哈哈，怎么样？代码参见 07_scaled_plot_large.html。但愿看到这些之后，你能意识到：如果用户说想要一个 800 像素而不是 600 像素宽的图，你就不用因为改代码而熬一个通宵了。没错，因为有了我（以及 D3 内置的聪明方法）你可以多睡一会了。身体是革命的本钱，记着我的好啊。

7.7 其他方法

`d3.scale.linear()` 还有几个非常方便的方法，有必要在这里简单介绍一下。

- `nice()`

  告诉比例尺取得为 `range()` 设置的任何值域，把两端的值扩展到最接近的整数。根据 D3 的维基："比如，值域 `[0.20147987687960267, 0.996679553296417]` 的优化值域为 `[0.2, 1]`。"这个方法对正常人都有用，因为人不是计算机，看到 0.20147987687960267 这样的数你一定会头大。

- `rangeRound()`

  用 `rangeRound()` 代替 `range()` 后，则比例尺输出的所有值都会舍入到最接近的整数值。对输出值取整有利于图形对应精确的像素值，避免边缘出现模糊不清的锯齿。

- `clamp()`

  默认情况下，线性比例尺可以返回指定范围之外的值。例如，假如给定的值位于输入值域之外，那么比例尺也会返回一个位于输出范围之外的值。不过，在比例尺上调用 `clamp(true)` 后，就可以强制所有输出值都位于指定的范围内。这意味着超出范围的值，会被取整到范围的最低值或最高值（总之是最接近的那个值）。

使用上述任何一个方法，只要把它们连缀到定义原始比例尺的方法链即可。比如，要使用 `nice()`，可以这样：

```
var scale = d3.scale.linear()
                    .domain([0.123, 4.567])
                    .range([0, 500])
                    .nice();
```

7.8 其他比例尺

除了（前面讨论的）线性（`linear`）比例尺，D3 还内置了另外几个比例尺方法。

- `sqrt`

  平方根比例尺。

- `pow`

  幂比例尺，适合值以指数级变化的数据集。

- `log`

  对数比例尺。

- `quantize`

  输出范围为独立的值的线性比例尺，适合想把数据分类的情形。

- `quantile`

  与 `quantize` 类似，但输入值域是独立的值，适合已经对数据分类的情形。

- `ordinal`

  使用非定量值（如类名）作为输出的序数比例尺，非常适合比较苹果和桔子。

- `d3.scale.category10()`、`d3.scale.category20()`、`d3.scale.category20b()` 和 `d3.scale.category20c()`

  能够输出 10 到 20 种类别颜色的预设序数比例尺，非常方便。

- `d3.time.scale()`

  针对日期和时间值的一个比例尺方法，可以对日期刻度作特殊处理。

  领略了比例尺的威力之后，该通过一些东西来表达它们了，通过什么呢？对，数轴！

第 8 章

数轴

掌握了 D3 比例尺的用法，我们制作出了如图 8-1 所示的散点图。



图 8-1：大型、成比例缩放的散点图

下面该添加水平和垂直的数轴了，而且也该把那些充斥于图表间的红色数字一笔勾销了。

8.1　数轴简介

与比例尺相似，D3 的数轴实际上也是由你来定义参数的函数。但与比例尺不同的是，调用数轴函数并不会返回值，而是会生成数轴相关的可见元素，包括轴线、标签和刻度。

但要注意，数轴函数只适用于 SVG 图形，因为它们生成的都是 SVG 元素。同样，数轴是设计与定量比例尺（与序数比例尺相对）配合使用的。

8.2 设定数轴

使用 d3.svg.axis() 可以创建通用的数轴函数：

```
var xAxis = d3.svg.axis();
```

要使用数轴，最起码要告诉它基于什么比例尺工作。在此，我们把绘制散点图时定义的 xScale 传给它：

```
xAxis.scale(xScale);
```

还可以继续设置标签相对数轴显示在什么地方。默认位置是底部，也就是标签会出现在轴线下方。（虽然是默认值，但明确指定也不会引起异常。）水平数轴的位置可以在顶部也可以在底部。而垂直数轴则要么在左要么在右：

```
xAxis.orient("bottom");
```

当然，把这些方法连缀在一行会更简洁：

```
var xAxis = d3.svg.axis()
                  .scale(xScale)
                  .orient("bottom");
```

最后，要想实际生成数轴并把那些线条和标签插入到 SVG 中，必须调用 xAxis 函数。这一点与使用比例尺函数类似，即要先配置一番（设置参数），然后再通过调用将其付诸实用。

我想把这些代码放到脚本底部，以便在 SVG 中的其他元素都生成之后再生成数轴，这样数轴就可以出现在“上面”了：

```
svg.append("g")
    .call(xAxis);
```

的确看着有点不那么舒服。你可能会问，为什么调用数轴函数和调用比例尺函数看起来那么不一样呢？听我解释：因为数轴函数实际上会在屏幕上绘制一些东西（通过把 SVG 元素添加到 DOM），所以我们需要指定在 DOM 的什么地方插入这些新元素。这显然跟比例尺函数不一样，比例尺（比如 xScale()），只是根据输入值来计算并返回值，主要是在其他函数里调用，不会影响 DOM。

好，前面的代码首先引用了 svg，即 DOM 中的 SVG 元素。然后，append() 在这个元素的末尾追加了一个新的 g 元素。在 SVG 标签内，g 元素就是一个分组

（group）元素。分组元素是不可见的，跟 line、rect 和 circle 不一样，但它有两大用途：一是可以用来包含（或“组织”）其他元素，好让代码看起来简洁整齐；二是可以对整个分组应用变换，从而影响到该组中所有元素（line、rect 和 circle）的视觉表现。关于变换，我们稍后就会介绍。

创建了新的 g 元素后，直接在这个元素上面调用了 call() 方法。那么 call() 有什么用呢？

D3 的 call() 函数会取得（比如刚才代码链中）传递过来的元素，然后再把它交给其他函数。对我们这例子而言，传递过来的元素就是新的分组元素 g（虽然这个元素不是必需的，但鉴于数轴函数需要生成很多线条和数值，有了它就可以把所有元素都封装在一个分组对象内）。而 call() 接着把 g 交给了 xAxis 函数，也就是要在 g 元素里面生成数轴。

假设你喜欢把代码写得让人不容易看懂，那么可以把前面的两段合成下面这一段：

```
svg.append("g")
    .call(d3.svg.axis()
    .scale(xScale)
    .orient("bottom"));
```

瞧，一行代码就定义并调用了数轴函数。不过，考虑到我们大脑的喜好，还是像前面那样先定义函数，然后再调用它更好理解。

无论如何，图 8-2 就是现在的结果。代码可以参考 01_axes.html。

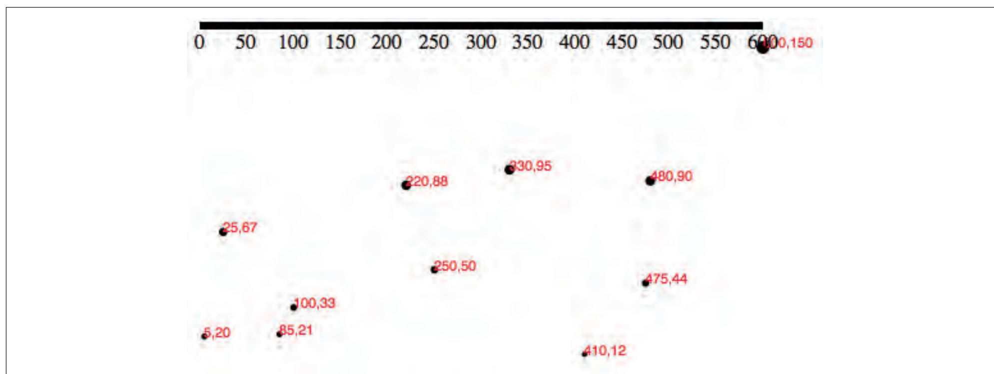


图 8-2：简单，但不好看的数轴

8.3 修整数轴

严格来讲，那确实是一个数轴。但从实用角度看，这家伙既不好看，也不中用。接

下来看看怎么给它打扮一下。先给新创建的 g 元素指定一个 axis 类吧，这样好给它添加 CSS 样式：

```
svg.append("g")
    .attr("class", "axis") // 指定 "axis" 类
    .call(xAxis);
```

然后，在 <head> 中的 <style> 标签里写两条 CSS 样式规则：

```
.axis path,
.axis line {
    fill: none;
    stroke: black;
    shape-rendering: crispEdges;
}

.axis text {
    font-family: sans-serif;
    font-size: 11px;
}
```

现在知道把所有数轴元素都组织在一个 g 分组中的好处了吧？这样只要使用简单的 CSS 选择符 .axis 就能为其中的任何元素应用样式。数轴本身是由 path、line 和 text 元素构成的，因此上面 CSS 瞄准了这三个元素。其中，路径（path）和线条（line）可以共用相同的规则，而文本（text）则有自己的字体和字号设置。

注意到了吗，通过 CSS 给 SVG 元素应用样式时，只能使用 SVG 的属性名，而不能使用常规的 CSS 属性。要注意的是，虽然很多属性在 CSS 和 SVG 中的名字相同，但也有一些不相同。比如，要使用常规的 CSS 属性设置文本颜色，那应该这样写：

```
p {
    color: olive;
}
```

这样就给所有段落 p 的文本设定了橄榄绿色（olive）。但同样的属性如果应用给 SVG 元素，比如

```
text {
    color: olive;
}
```

不会有什么效果，因为 color 不是 SVG 能够识别的属性名。这时候，必须使用 SVG 中不同名但作用一样的 fill 属性：

```
text {
    fill: olive;
}
```

如果你在给 SVG 元素应用样式时，发现 CSS 代码根本不起作用，我劝你不要心急。

深呼吸，稍等一下，再仔细瞧瞧那些属性名，一定要保证是 SVG 的，而不是 CSS 的。（要了解 SVG 都有哪些属性，可以参考 MDN 网站：https://developer.mozilla.org/en-US/docs/SVG/Attribute。）

这里用到的 `shape-rendering` 也是一个奇怪的 SVG 属性。使用它是为了保证数轴和刻度线精确到像素级。不要给我一个模糊的数轴！

用 CSS 打扮后的数轴如图 8-3 所示。



图 8-3：清爽一些的数轴

好些了，但数轴顶部的线被切掉了，而且数轴本来应该显示在图表底下嘛。这时候，就该用到 SVG 变换（transform）了。只要添加一行代码，就可以把整个数轴分组平移到图表下方：

```
svg.append("g")
    .attr("class", "axis")
    .attr("transform", "translate(0," + (h - padding) + ")")
    .call(xAxis);
```

新增的这行代码在 `attr()` 中设置了 `g` 元素的属性 `transform`。SVG 中的变换功能非常强大，有多种不同的变换方式，包括缩放和旋转。但我们暂时只介绍平移（translation）变换，它可以把整个 `g` 分组向下移动一定距离。

平移变换的语法很简单，就是 `translate(x,y)`，其中 `x` 和 `y` 的含义都非常明确，就是要把元素移动到的新位置的 *x* 和 *y* 坐标。因此，在 DOM 中我们会看到 `g` 元素动态添加了如下代码：

```
<g class="axis" transform="translate(0,280)">
```

由此可见，`g.axis` 不会水平移动，但会向下移动 280 像素，正好移到图表底部。下面大概解释一下设置平移变换的代码：

```
        .attr("transform", "translate(0," + (h - padding) + ")")
```

注意这里使用了（h-padding），这是要把分组的顶边 y 坐标设置为 h，即整个 SVG 元素的高度——然后，再减去我们前面定义的边距值（padding）。经过计算，（h-padding）等于 280，再跟其他字符串拼接起来，就得到了最终的变换属性值：translate(0,280)。

图 8-4 所示的结果看起来舒服多了！完整代码请查看 02_axes_bottom.html 吧。

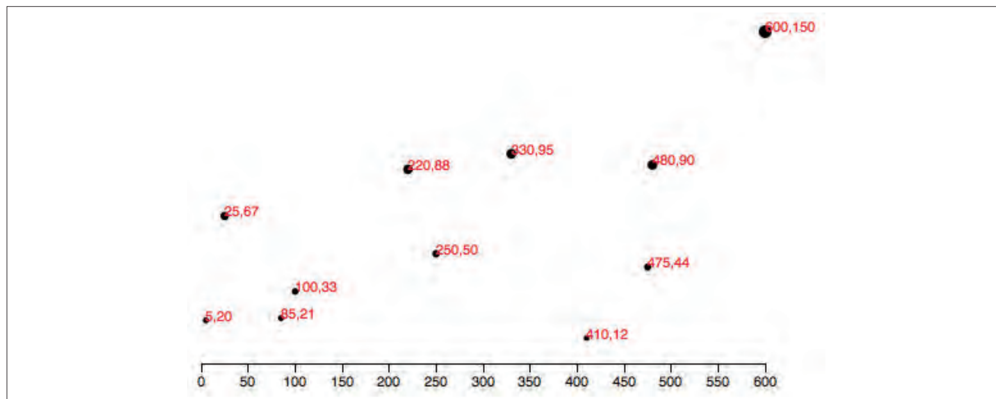


图 8-4：清爽怡人的数轴来了

8.4　优化刻度

数轴的刻度线（tick）是用来传达信息的，但也不是越多越好，多过某个数量，刻度线反而会让图表显得混乱。到现在为止，我们还没有指定数轴上要显示多少根刻度线，也没有指定两根刻度线的间距。就在我们全无指示的情况下，D3 自动检测了比例尺 xScale，从而作出了有数量依据的判断，帮我们确定了要画多少刻度线，以及刻度线之间的间隔（这里是 50 像素）。

当然，你可以干预数轴的任何方面。比如，使用 ticks() 方法就可以粗略地指定刻度线的数量：

```
var xAxis = d3.svg.axis()
                  .scale(xScale)
                  .orient("bottom")
                  .ticks(5);  // 粗略地设置刻度线的数量
```

完整代码请参见 03_axes_clean.html。

通过图 8-5 可以看到，尽管我们设置了 5 根刻度线，但 D3 则擅自作主，画出了 7 根。这是 D3 在保护你呢，因为它发现 5 根刻度线会把输入值域切分成不够恰当的

值（也就是 0、150、300、450 和 600）。D3 只将 `ticks()` 的值作为一个建议，如果它发现有更清晰更方便理解的值（比如以 100 为间隔），哪怕比你设置的值多一点或少一点，它也会采用。实际上，这是一个非常出彩的功能，可以确保设计的可伸缩性。随着数据集的变化，输入值域的扩大或缩小（数值变得更大或更小），D3 都可以保证刻度标签的数量合适而且易读。

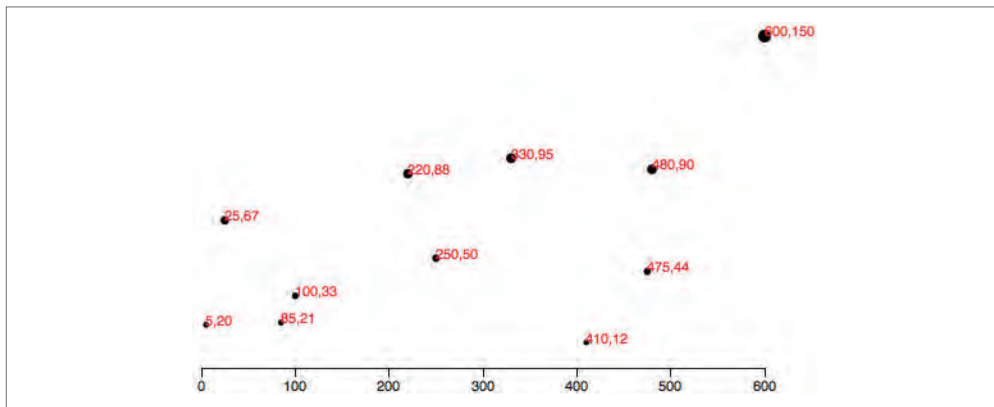


图 8-5：刻度线变少了

8.5　垂直数轴

接下来该考虑一下垂直方向的数轴了！通过复制修改为 `xAxis` 而写的代码，可以在代码的顶部添加如下几行：

```
// 定义 y 轴
var yAxis = d3.svg.axis()
                  .scale(yScale)
                  .orient("left")
                  .ticks(5);
```

而在代码底部添加如下几行：

```
 // 创建 y 轴
svg.append("g")
    .attr("class", "axis")
    .attr("transform", "translate(" + padding + ",0)")
    .call(yAxis);
```

看看图 8-6，*y* 轴的标签左对齐，而 `yAxis` 的分组 `g` 向右平移了预先设定的边距。

现在看起来像是个真正的图表了！但 `yAxis` 的标签被切掉了一些。为了增大标签左侧的空间，我们把边距的值由 20 加到 30：

```
var padding = 30;
```

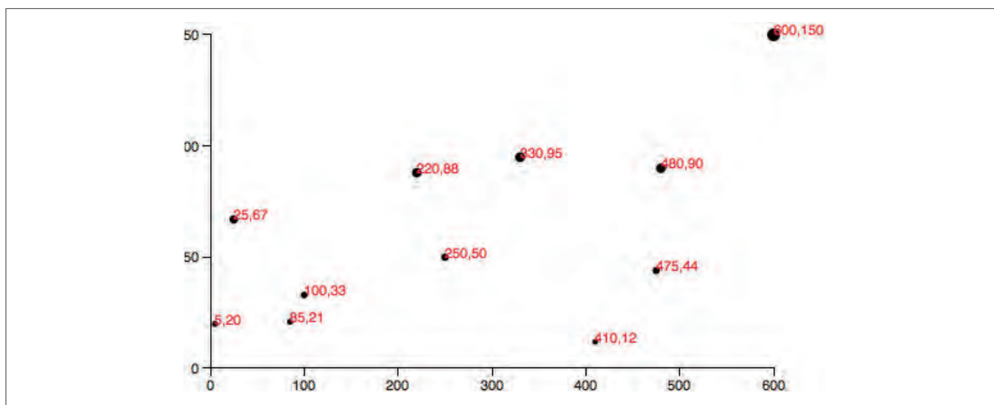


图 8-6：初始状态下的 *y* 轴

当然，对于不同的数轴也可以分别声明不同的边距，比如可以分别命名为 `xPadding` 和 `yPadding`。这样就能更细致地控制布局了。

更新后的代码参见 04_axes_y.html，而结果如图 8-7 所示。

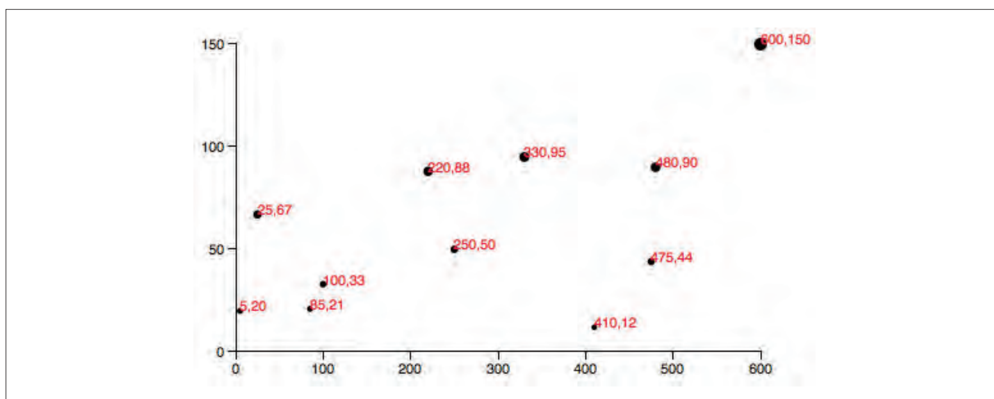


图 8-7：添加 *y* 轴后的散点图

8.6　最后的润色

非常感谢各位到目前为止保持克制和礼貌，一点都没有嫌烦。当然，我也一直在努力不让你讨厌。接下来，为了证明新坐标轴是动态可伸缩的，我们把静态的数据集改成随机生成的：

```
// 动态的随机生成的数据集
var dataset = [];
var numDataPoints = 50;
```

```
var xRange = Math.random() * 1000;
var yRange = Math.random() * 1000;
for (var i = 0; i < numDataPoints; i++) {
    var newNumber1 = Math.floor(Math.random() * xRange);
    var newNumber2 = Math.floor(Math.random() * yRange);
    dataset.push([newNumber1, newNumber2]);
}
```

这里首先初始化了一个空数组，然后循环 50 次，每次选择两个随机数值，添加（`push()`）到数据集的数组中。结果如图 8-8 所示。

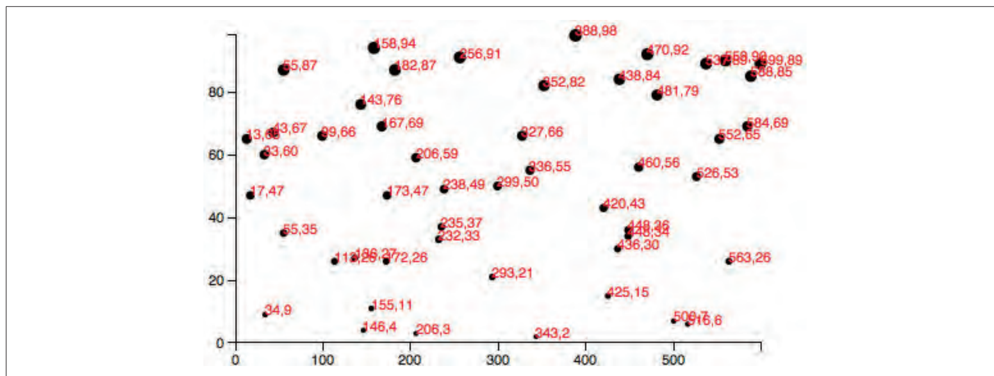


图 8-8：根据随机数据生成的散点图

试试使用随机数据的示例 05_axes_random.html。每次刷新页面，都会生成不同的数据集。注意，数轴会随着输入值域的变化而相应地缩放，而刻度和标签也会相应地变化。

讲完了数轴，我们该把那些讨厌的红色标签去掉了。很简单，只要把相应的代码注释掉即可。

最终结果如图 8-9 所示，而完成的散点图代码则在 06_axes_no_labels.html 中。

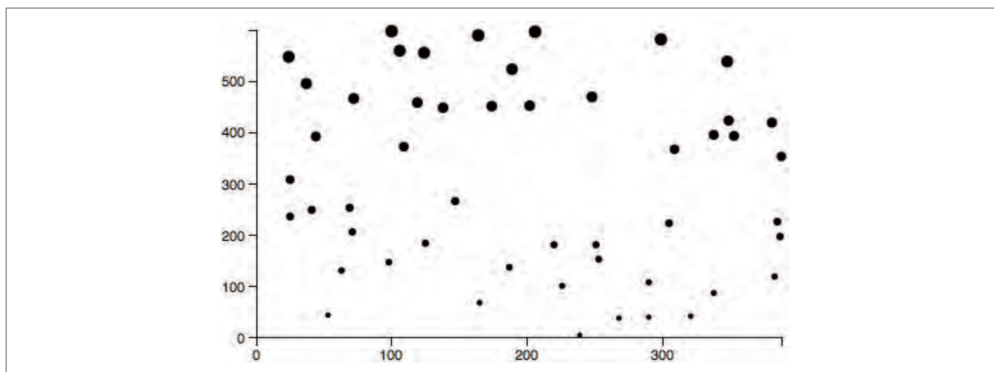


图 8-9：根据随机数据生成的散点图，去掉了红色标签

8.7　为刻度标签定义样式

迄今为止，我们的数据集中都是整数，这当然是最简单的情况。但实际应用中的数据可没有那么简单，到时候你就会想要对数轴标签进行更多控制，包括数字的格式。好了，使用 `tickFormat()` 就可以为数值应用不同的格式。比如，让数值保留小数点后三位数字，或者显示为百分比值，或者同时应用这两种格式。

要使用 `tickFormat()`，首先要定义一个新的数值格式函数。通过这个函数可以告诉 D3 把数值当成百分比，同时保留一位小数，等等。这样，交给它数值 0.23，它就会返回字符串 `"23.0%"`。（关于 `d3.format()` 的更多设置，请参考这里：https://github.com/mbostock/d3/wiki/Formatting#wiki-d3_format。）

```
var formatAsPercentage = d3.format(".1%");
```

然后，让数轴对其刻度使用刚刚定义的格式化函数，比如：

```
xAxis.tickFormat(formatAsPercentage);
```



要测试这几个函数，其实最简单的办法就是使用控制台。比如，随便打开一个加载 D3 的页面，比如 06_axes_no_labels.html，在控制台里直接输入格式化规则，然后给它一个值，就像你给任何函数传递参数一样。

在图 8-10 中可以看到，输入的数值 0.54321 被转换成了适合显示的 54.3%——完美！

```
> var formatAsPercentage = d3.format(".1%");
  undefined
> formatAsPercentage(0.54321)
  "54.3%"
>
```

图 8-10：在控制台中测试 `format()` 函数

试试 07_axes_format.html 吧。很显然，百分比格式并不适合当前数据集之下的散点图。但这只不过是一次练习嘛，你可以尝试生成非整数的随机数，或者直接拿格式化函数试验一下。

第 9 章

更新、过渡和动画

目前为止，我们只使用了静态的数据集。但现实中的数据总是在随着时间变化，而我们很可能需要及时将这些变化反映到图表中。

用 D3 的话来说，这些变化要通过*更新*来处理。视觉上的调整以*过渡*的形式展现，而过渡则利用了*动画*这个视觉利器。

本书的例子将首先根据一个数据集来生成图表，然后再彻底改变数据。

9.1 更新条形图

先来看看我们的老朋友，图 9-1 所示的条形图。

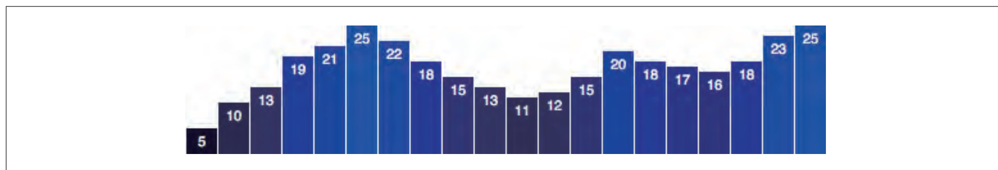


图 9-1：条形图，老朋友了

看一看 01_bar_chart.html 中的代码，就会发现这张图用的是静态数据集：

```
var dataset = [ 5, 10, 13, 19, 21, 25, 22, 18, 15, 13,
                11, 12, 15, 20, 18, 17, 16, 18, 23, 25 ];
```

在此基础上，我们又学习了如何编写更灵活的代码，让图表元素能够自动调整以适

应不同规模的数据集（或长或短的数组），以及不同的数据值（或小或大的数值）。D3 的比例尺让我们实现了这种灵活控制，因此接下来我们把这个条形图也更新一番，让它跟上时代。

准备好啦？好，稍等一会…… 嗯……完了！让您久等了。

图 9-2 看起来虽然类似，但内部已经有了很大的变化。不信可以打开 02_bar_chart_with_scales.html 看看。

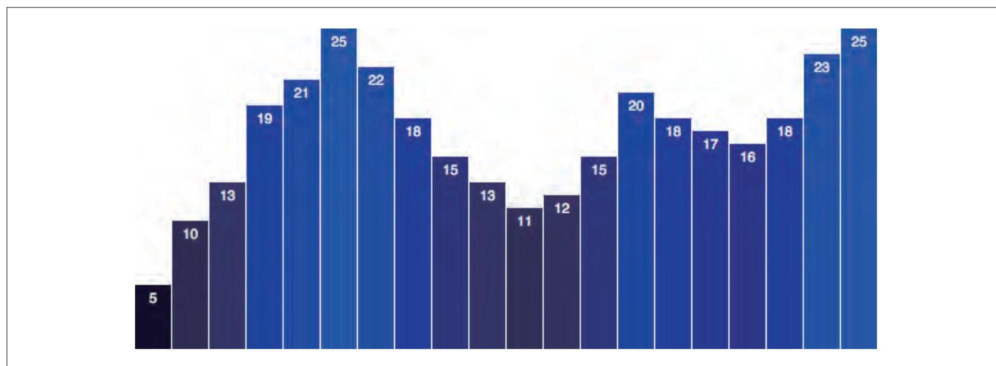


图 9-2：焕然一新的可伸缩的灵活的条形图

一上来，我们先调整了宽度和高度，让条形图更高一些也更宽一些：

```
var w = 600;
var h = 250;
```

接下来定义了一个序数比例尺，用于处理条形左 / 右定位和 x 轴的标签：

```
var xScale = d3.scale.ordinal()
                .domain(d3.range(dataset.length))
                .rangeRoundBands([0, w], 0.05);
```

看起来可不好理解啊，没关系，我们一行一行解释。

9.1.1　序数比例尺

首先，第一行：

```
var xScale = d3.scale.ordinal()
```

声明了一个变量 xScale，跟散点图示例中一样。只不过这里是序数比例尺，而不是线性比例尺。什么是序数比例尺？就是用于处理序数的嘛。那什么是序数呢？序数就是有固定顺序的一些类别，比如：

- 新生、大二、大三、毕业班
- B 等、A 等、AA 等
- 非常不喜欢、不喜欢、没感觉、喜欢、非常喜欢

当然，这个条形图使用的也不是真正的序数数据集，而只是从左到右地原样表现了相应值在数据集中的顺序。D3 的序数比例尺对这种情况也适用，因为它有很多可见的元素（竖条），以一定的顺序展示（从左到右），而且均匀分布。这一点稍后你就会明白。

```
.domain(d3.range(dataset.length))
```

第二行代码为比例尺设定了输入值的值域。还记得吗，对线性比例尺可是要用包含两个值的数组来设置值域的，比如 `[0, 100]`。而且在线性比例尺中，这个数组包含的必须是值域的最小值和最大值。而序数的值域呢，就是序数，不是线性或定量数据。因此设置序数比例尺的值域，通常是要指定一个包含类别名称的数组，比如：

```
.domain(["新生", "大二", "大三", "毕业班"])
```

我们的条形图没有明确的类别，但可以给每个数据点或条形分配一个对应其在数据集中位置的 ID 值，比如 0、1、2、3，等等。因此，相应的值域可以这样设置：

```
.domain([0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
         10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

不过，在此我们可以使用一个更好的生成连续数值数组的方法：`d3.range()`。

在查看 02_bar_chart_with_scales.html 的时候，打开控制台，输入以下代码：

```
d3.range(10)
```

应该可以看到它输出了如下数组：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这多好啊。D3 又给我们省时间了（也省掉了使用 `for()` 循环的麻烦）。

回到我们的代码中，现在该明白这行代码的含义了吧：

```
.domain(d3.range(dataset.length))
```

1. 这里的 `dataset.length` 等于 20，因为数据集里一共有 20 个值；
2. 那当然就会调用 `d3.range(20)`，调用结果就是返回这个数组：`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]`；
3. 最后，`domain()` 把新序数比例尺的值域设置为这些值。

比较容易让人迷惑的地方在哪儿呢？因为序数通常是非数值的，而我们这里使用的却是数值（0、1、2……）。

9.1.2 自动分档

`d3.scale.ordinal()` 有一个优势，它支持范围分档（banding）。与定量比例尺（如 `d3.scale.linear()`）返回连续的范围值不同，序数比例尺使用的是离散范围值，也就是输出值是事先就确定好的，可以是数值，也可以不是。

映射范围时，可以使用 `range()`，也可以使用 `rangeBands()`。后者接收一个最小值和一个最大值，然后根据输入值域的长度自动将其切分成相等的块或“档”，例如：

```
.rangeBands([0, w])
```

这行代码的意思是“计算从 0 到 w 可以均分为几档，然后把比例尺的范围设定为这些档”。就我们的例子而言，值域有 20 个值，因此 D3 会执行如下计算：

```
(w - 0) / xScale.domain().length
(600 - 0) / 20
600 / 20
30
```

最后，每一档的“宽度”为 30。

可以给 `rangeBands()` 传入第二个参数，指定档间距。在此，我们使用的是 0.2，也就是档间距为每一档宽度的 20%：

```
.rangeBands([0, w], 0.2)
```

要想得到整数值，可以使用 `rangeRoundBands()`，它与 `rangeBands()` 的区别只是输出的值会舍入为最接近的整数。因此，举个例子，12.3456 就会变成 12。整数值有助于将视觉元素与像素网格对齐，保证图形的边缘清晰锐利。

同时，我们还把条形间距减少到 5%，因此那行代码最后应该如下所示：

```
.rangeRoundBands([0, w], 0.05);
```

这样可以保证表示条形宽度的像素值恰如其分，而且条形间还会有很小的间距。

9.1.3 使用序数比例尺

在前面的代码中（建议大家看一看源代码），创建 `rect` 元素的时候还设定了图表水平的 *x* 轴坐标：

```
// 创建条形
svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   .attr("x", function(d, i) {
      return xScale(i);          // <-- 设定 x 坐标
   })
            ...
```

这里请注意，因为匿名函数的参数是 d 和 i，所以 D3 会自动传入正确的值。当然，d 还是当前的数据值，而 i 是它的索引值。也就是说，i 在每次循环中分别会接收到 0、1、2、3……

多巧啊！设置比例尺值域的时候，我们使用的也是同样的值（0、1、2、3……）。因此，在调用 xScale(i) 时，xScale() 会找到序数值 i，然后返回对应的输出（档位）值。（你可以自己在控制台中验证一下，输入 xScale(0) 或 xScale(5) 试试。）

更方便的是设置条形的宽度变得很容易。在使用序数比例尺之前，我们必须这样：

```
.attr("width", w / dataset.length - barPadding)
```

因为 rangeRoundBands() 已经内置了间距，现在连 barPadding 也用不着了。设置每个条形的宽度，可以直接使用以下代码：

```
.attr("width", xScale.rangeBand())
```

D3 帮我们做算术的感觉怎么样？

9.1.4 其他更新

在 02_bar_chart_with_scales.html 里，我还更新了其他一些地方，包括创建了新的线性比例尺 yScale ，以计算垂直方向的值。关于线性比例尺，你已经知道怎么用了，自己看代码时只要关注怎么设置条形的高度就好了。

9.2 更新数据

好，如图 9-3 所示，我们现在有了令人惊叹的条形图，它足够灵活，能够处理任何数据。

什么，你说真的？让我们看看。

最简单的数据变更的情况是所有值都不一样了，但值的数量没有变。

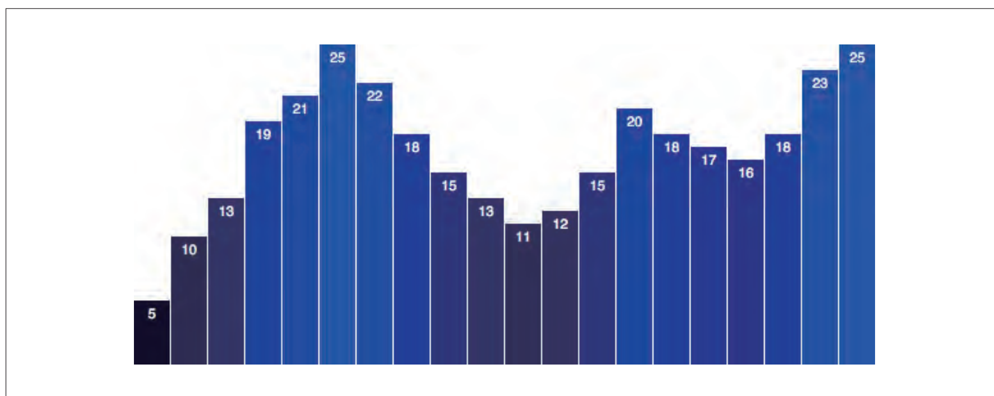


图 9-3：新条形图

在这种情况下，可以这样做：

1. 修改数据集的值；
2. 把新值重新绑定到已有元素（这样才能重写原来的值）；
3. 按需设置新属性值，以更新可见的元素。

不过，在实践这几个步骤之前，需要触发某个事件。到目前为止，我们的代码都是随着页面加载执行。对于更新数据来说，可以在开始的绘制代码一执行完毕就更新，但这样更新就不为人知了，因为速度太快。为了能看到更新的变化，我们要把更新的代码与其他代码分开。因此，需要在页面加载之后有一个“触发器”，以触发数据和图表的更新。鼠标点击怎么样？

9.2.1 通过事件监听器实现交互

要响应鼠标点击，任何 DOM 元素都可以胜任。这次就不用按钮了，直接在 HTML 的主体中添加一个 p 标签就好：

```
<p>Click on this text to update the chart with new data values (once).</p>
```

然后在 D3 代码的最后，添加如下几行：

```
d3.select("p")
    .on("click", function() {
        // 被单击时执行任务
    });
```

这样就选中了新创建的 p 元素，然后为它添加了一个事件监听器。什么是事件监听器啊？

在 JavaScript 中，事件随时都在发生。这里所说的事件，可不是连篇累牍的新闻报道，而是 mouseover 和 click。大多数情况下，这些至关重要的事件都会被忽略（现实生活中差不多也这样，对吧？）。可如果有人监听，那就可以听到这些事件，而且能够把它传给自己的子孙后代，好啦，至少可以触发某种 DOM 交互。（关于这一点，有一桩经典的 JavaScript 公案：如果有事件发生，而没有监听器听到，那它到底发生过没有？）

什么是事件监听器呢，其实也是匿名函数，可以监听（一或多个）特定元素上发生的事件。D3 的 selection.on() 方法是添加事件监听器的简便方法，它接受两个参数：事件类型（"click"）和监听器（匿名函数）。

我们需要监听器监听在选择的 p 元素上发生的单击事件。在该事件发生时，就会执行监听函数。而在匿名监听函数的花括号内，可以编写任何需要的代码：

```
d3.select("p")
    .on("click", function() {
        // 响应单击
        alert("Hey, don't click that!");
    });
```

第 10 章还会进一步讨论交互。

9.2.2 改变数据

事实上，我们不想弹出那个令人讨厌的窗口，而是想在监听器中更新数据集，覆盖原始的数值。这是一开始我们说的第 1 步：

```
dataset = [ 11, 12, 15, 20, 18, 17, 16, 18, 23, 25,
            5, 10, 13, 19, 21, 25, 22, 18, 15, 13 ];
```

第 2 步是重新绑定新值与已有元素。为此，需要选择相应的矩形，再调用一次 data() 方法：

```
svg.selectAll("rect")
   .data(dataset);     // 长官，新数据绑定成功！
```

9.2.3 更新视觉元素

最后，第 3 步是更新视觉元素的属性，以反映（刚刚更新的）数据值。这一步超简单，因为只把原来写过的代码复制粘贴过来就行了。此时此刻，所有矩形的水平位置和宽度都不变，需要更新的只有它们的高度和 y 坐标值，所以我们需要以下代码：

```
svg.selectAll("rect")
```

```
        .data(dataset)
        .attr("y", function(d) {
             return h - yScale(d);
        })
        .attr("height", function(d) {
             return yScale(d);
        });
```

看到了吧，除了没有 enter() 和 append()，所有代码跟最初生成矩形的代码完全一样。

把这三步的所有代码放到一起，就是更新所需的代码：

```
    //单击的时候，更新数据
    d3.select("p")
        .on("click", function() {

            //新数据集
            dataset = [ 11, 12, 15, 20, 18, 17, 16, 18, 23, 25,
                        5, 10, 13, 19, 21, 25, 22, 18, 15, 13 ];

            //更新所有矩形
            svg.selectAll("rect")
               .data(dataset)
               .attr("y", function(d) {
                    return h - yScale(d);
               })
               .attr("height", function(d) {
                    return yScale(d);
               });

        });
```

看看 03_updates_all_data.html 吧，开始状态如图 9-4 所示。

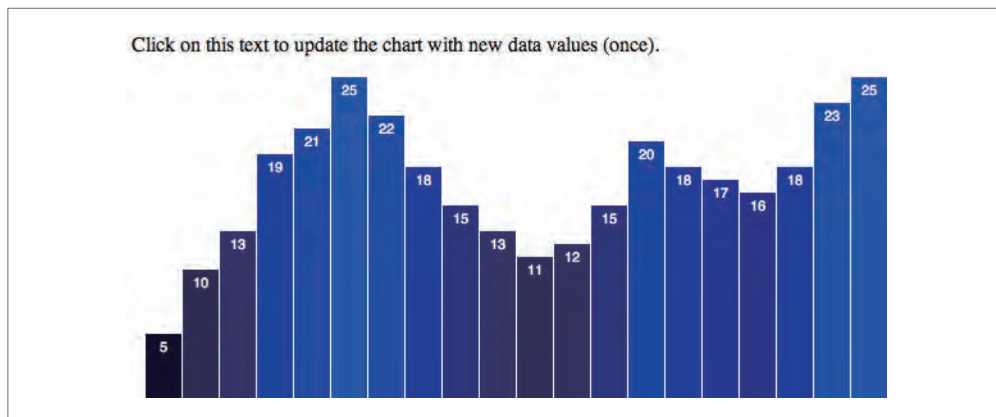


图 9-4：可更新的条形图

然后单击段落中的任何地方，图表就会变成图 9-5 所示。

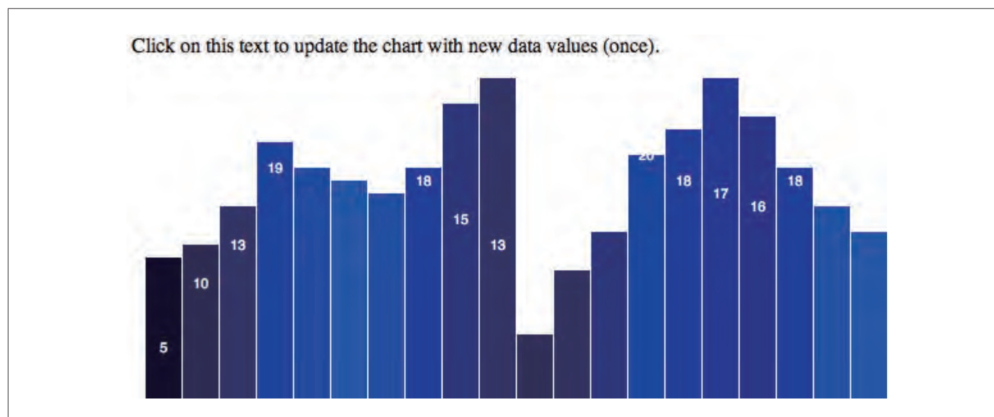


图 9-5：更新数据后的条形图

好消息：数据集中的值全都改变、重新绑定并反映在矩形上面了。坏消息：条形图看起来令人匪夷所思，我们忘了更新标签，还有条形的颜色了。还有一个好消息（我总喜欢把最好的消息放在最后公布）：这个问题容易改。

为了保证颜色也同步更新，只要把原来针对 fill 编写的代码复制过来就行：

```
svg.selectAll("rect")
   .data(dataset)
   .attr("y", function(d) {
        return h - yScale(d);
   })
   .attr("height", function(d) {
        return yScale(d);
   })
   .attr("fill", function(d) {   // <-- 在这儿呢!
        return "rgb(0, 0, " + (d * 10) + ")";
   });
```

接下来更新标签，方法类似，只是这里要调整一下它们的文本内容和 *x*、*y* 坐标值：

```
svg.selectAll("text")
   .data(dataset)
   .text(function(d) {
        return d;
   })
   .attr("x", function(d, i) {
        return xScale(i) + xScale.rangeBand() / 2;
   })
   .attr("y", function(d) {
        return h - yScale(d) + 14;
   });
```

好啦，打开 04_updates_all_data_fixed.html 再看看吧。你会发现一开始跟最初一样，不过单击段落之后，就能看到正确更新后的条形颜色和标签了，如图 9-6 所示。

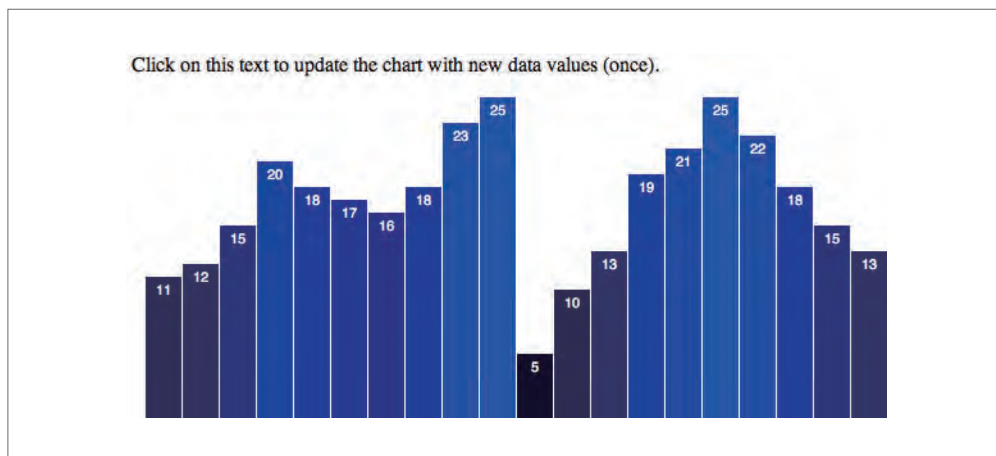


图 9-6：更新了条形图的颜色和标签

9.3 过渡动画

人生从一种境遇到另一种境遇的过渡可谓惊险，第一天上学，搬到新城市，辞掉全职工作去干自由数据可视化。但 D3 的过渡却好玩、惊艳，而且没有任何精神负担。

要创建一个精致、流畅、动态的过渡效果，只需简单的一行代码：

```
.transition()
```

特别要注意的是，在方法链上，要把这个调用插到选择元素之后，改变任何属性之前：

```
//更新所有矩形
svg.selectAll("rect")
   .data(dataset)
   .transition()    // <-- 这是新代码，其他都保持不变。
   .attr("y", function(d) {
        return h - yScale(d);
   })
   .attr("height", function(d) {
        return yScale(d);
   })
   .attr("fill", function(d) {
        return "rgb(0, 0, " + (d * 10) + ")";
   });
```

好啦，现在打开 05_transition.html，然后点击段落看看过渡动画是什么效果吧。更新后的最终效果相同，但从初始状态到最终状态的过渡则自然、顺畅多了。

是不是太疯狂了？我不是心理学家，可一行代码就能让 D3 帮我们实现动画过渡，简直太疯狂了。

没有 `transition()`，D3 会对所有 `attr()` 语句立即求值，因此高度和填充的变化立刻会发生。添加了 `transition()` 之后，D3 会考虑时间的因素。这一次，它不再一次性应用所有新值，而是会在旧值与新值之间插入一系列过渡值，也就是要对起始和终止值进行归一化处理，然后计算两者之间的中间状态。D3 还能分辨不同属性值的格式，对这些格式的变化进行插值。比如，高度从 `200px` 开始过渡到 `100`（没有 `px`），或者把蓝色填充变成 `rgb(0,255,0)`（绿色）。整个过程不需要你参与，D3 就替你做好了。

你还不相信我？不过，这确实有点疯狂。当然，也超级好用。

9.3.1 持续时间

对，D3 会随着时间推移不断插入 `attr()` 值，但持续多长时间呢？默认是 250 毫秒，或四分之一秒（1 秒等于 1000 毫秒）。这就是 05_transition.html 中的动画那么快的原因。

好在，我们可以控制动画的持续时间。而且，只要添加一行代码（这次同样不是开玩笑）：

```
.duration(1000)
```

注意，必须把 `duration()` 放到 `transition()` 之后，参数的单位是毫秒，因此 `duration(1000)` 就是持续 1 秒。

下面把这行代码放到页面中：

```
// 更新所有矩形
svg.selectAll("rect")
   .data(dataset)
   .transition()
   .duration(1000)  // <-- 这是新代码!
   .attr("y", function(d) {
        return h - yScale(d);
   })
   .attr("height", function(d) {
        return yScale(d);
```

```
        })
        .attr("fill", function(d) {
             return "rgb(0, 0, " + (d * 10) + ")";
        });
```

打开 06_duration.html 看看有什么不一样。复制这个文件，然后把持续时间改成其他值，观察不同的效果。比如，改成 `3000`（3 秒）或 `100`（十分之一秒）。

实际的持续时间取决于图表的类型，以及触发过渡的方式。根据经验，细微的界面反馈（比如鼠标悬停在元素上触发过渡），过渡时间大约 150 毫秒比较合适，而更显著的视觉过渡（比如整个数据视图的变化）持续 1000 毫秒比较理想。1000 毫秒（1 秒）不算长，也不算短。

如果你自己懒得动手试验，可以直接打开 07_duration_slow.html，我在其中设定了 `5000` 这个值，让过度动画持续 5 秒。

在这个慢镜头之下，可以看到值标签并没有像条形高度那么平滑地过渡。要校正这个问题，也只需两行代码，这次新代码要插入到更新标签的代码段中：

```
    //更新标签
    svg.selectAll("text")
       .data(dataset)
       .transition()         // <-- 这是新代码
       .duration(5000)       //     这也是新代码
       .text(function(d) {
            return d;
       })
       .attr("x", function(d, i) {
            return xScale(i) + xScale.rangeBand() / 2;
       })
       .attr("y", function(d) {
            return h - yScale(d) + 14;
       });
```

好多啦！看看 08_duration_slow_labels_fixed.html 吧，现在标签和条形的变化已经很协调了。

9.3.2 缓动函数

5000 毫秒，慢得像糖稀一样的过渡，让我们感受到了整个动画的*品质*。注意到没，动画一开始非常慢，然后逐渐加速，最后在达到预定高度之前速度再次慢了下来。换句话说，动画的速度不是*线性不变*的，而是有加减速变化的。

应用于过渡效果的这种动画品质叫做*缓动*。用动画术语来说，可以理解成元素缓缓就位，然后从一个位置不间断地移动到另一个位置。

在 D3 中，可以使用 ease() 指定不同的缓动类型。默认的缓动效果是 "cubic-in-out"，产生的就是我们刚刚看到的那种逐渐加速然后再逐渐减速的效果。这种默认效果适合大多数平滑过渡。

对比一下 09_ease_linear.html，这个页面使用了 ease("linear") 指定了线性缓动函数。线性缓动就是没有逐渐加速和减速的变化，所有元素都按照一个速度变化，变化到最终值时戛然而止。（这个页面中动画的持续时间是 2000 毫秒。）

同样，也要在 transition() 之后、attr() 之前指定 ease()。事实上，ease() 在 duration() 之前之后都没问题，但先过渡再设置缓动似乎更顺理成章：

```
...    // 选择元素的代码
.transition()
.duration(2000)
.ease("linear")
...    //attr() 的代码
```

除此之外，还有很多缓动函数可供选择。以下是几个我比较喜欢的。

- circle
  逐渐进入并加速，然后突然停止。

- elastic
  描述这个效果的一个最恰当的词是“有弹性”。

- bounce
  像皮球落地一样反复弹跳，慢慢停下来。

这几种缓动的效果可以参见 10_ease_circle.html、11_ease_elastic.html 和 12_ease_bounce.html。要了解 D3 支持的所有缓动效果，可以参考它的维基：https://github.com/mbostock/d3/wiki/Transitions#wiki-d3_ease。

哇噢，我有点后悔告诉你弹跳的 bounce 了。请一定在想通过信息图来体现讽刺和挖苦的意思时使用 bounce（比如嘲笑别人的信息图做得有多差劲）。从直觉上说，我还没发现什么可视化图表适合使用弹跳动画。cubic-in-out 作为默认效果是有道理的。

9.3.3 延迟时间

与 ease() 控制动画品质不同，delay() 用于指定过渡什么时间开始。

可以给 delay() 传入一个静态的值，同样以毫秒为单位，比如：

```
...
.transition()
.delay(1000)      //1000 毫秒，即 1 秒
.duration(2000)   //2000 毫秒，即 2 秒
...
```

与使用 duration() 和 ease() 一样，把 delay() 放到哪里并没有十分严格的限制，但我更喜欢把它放在 duration() 前面。因为是先设定延迟时间，然后过渡动画才开始计时嘛，这符合逻辑。

参见 13_delay_static.html，打开后单击文本会触发 1000 毫秒的延迟（1 秒内什么也不会变化），然后才是 2000 毫秒的过渡动画。

静态延迟时间只是一种延迟方式，更有意思的是可以动态计算延迟时间。动态延迟的一个常见用途就是创建交错延迟的效果，让某些过渡在另一个过渡之前发生。交错延迟对人的感知有利，因为当相邻元素的变化不那么同步时，人眼更容易注意到每个元素的变化。要设置动态延迟，就别给 delay() 传递静态值，而是给它传入一个函数，按照 D3 的惯例……对，就是传入一个匿名函数：

```
...
.transition()
.delay(function(d, i) {
    return i * 100;
})
.duration(500)
...
```

在匿名函数中，与当前元素绑定的数据值是以 d 传入的，而这个元素的位置是以 i 传入的。因此，这里的意思是让 D3 循环遍历每个元素，把它们的动画延迟时间设定为 i * 100，也就是后一个元素的动画开始时间总比前一个元素晚 100 毫秒。

总之，这样就能得到一个交错过渡的动画。还是打开 14_delay_dynamic.html 亲自体验一下吧。

注意，我把持续时间缩短到了 500 毫秒，以便让整体动画效果显得更紧凑。而且，此时的 duration() 设定的是每个元素的动画持续时间，不是整体动画累计的持续时间。对这个例子来说，20 个元素的持续时间都是 500 毫秒，假如没有延迟，那所有元素都会用 500 毫秒（即半秒）完成动画。但如果依次给每个元素设定了 100 毫秒的延迟（i * 100），那么所有过渡动画的总时间将为 2400 毫秒：

```
i 的最大值乘以 100 毫秒延迟加上 500 毫秒持续时间 =
19 * 100 + 500 =
2400
```

由于这种延迟会逐个元素累积，所以数据值越多，全部过渡动画运行的总时间就越长。如果需要动态加载不同长度的数据集，那就要把这个因素考虑进去。假设数据集里一下子有了 10 000 个值（而不是 20 个），那要看完整段动画就得花很长时间（精确地说，要花 1 000 400 毫秒或 16.67 分钟）。一下子就不可爱了。

好在，我们可以根据数据集的长度动态调整延迟时间。对 D3 来说这没有什么大惊小怪的，只不过是一些数学计算而已。

打开 15_delay_dynamic_scaled.html 看一看，这里的数据集有 30 个值。如果你拿着秒表计时，会发现总过渡时间是 1.5 秒，或大约 1500 毫秒。

现在再看看 16_delay_dynamic_scaled_fewer.html，过渡代码一样，但只有 10 个数据点，而总过渡时间稍长了一些（大概长了 20%），因此总时间还是：1.5 秒！

怎么会这样呢？

```
...
.transition()
.delay(function(d, i) {
    return i / dataset.length * 1000;   // <-- 这里的代码是关键
})
.duration(500)
...
```

前面两个示例页面使用的都是这个延迟计算方法：把简单的用 `i` 乘以静态延迟时间值，改为先用 `dataset.length` 除 `i`（从而达到了把这个值归一化的效果），然后再用得到的值乘以 1000（即 1 秒）。结果就是最后一个元素的延迟时间为 1000，而之前每个元素的延迟时间渐次缩短。最大的延迟 1000 加上 500 等于 1.5 秒的总过渡时间。

这样来设置延迟非常合理，因为能确保代码的可伸缩性。无论数据点只有 10 个，还是多达 10 000 个，总的过渡时间都是可以接受的。

9.3.4 使用随机数据

为了展示这件事有多酷，我们要重用一下第 5 章的随机数生成代码。这样就能想更新多少次就更新多少次，而且每次使用的都是新数据集。找到响应单击事件的匿名函数，把其中的静态数据集替换成以下随机生成数据集的代码：

```
//生成新数据集
var numValues = dataset.length;                         //取得原数据集的长度
dataset = [];                                           //初始化空数组
for (var i = 0; i < numValues; i++) {                   //循环 numValues 次
```

```
        var newNumber = Math.floor(Math.random() * 25);   //生成新随机数（0~24）
        dataset.push(newNumber);                          //把新数值添加到数组
    }
```

这样就可以用随机生成的包含随机数（0~24）的数组，替换原来的静态数据集。新数组的长度与原数据集长度相同。

然后，更新一下段落的内容，给大家一点提示：

```
<p>Click on this text to update the chart with new data values as many
times as you like!</p>
```

现在打开 17_randomized_data.html 看看吧，应该能看到如图 9-7 所示的结果。

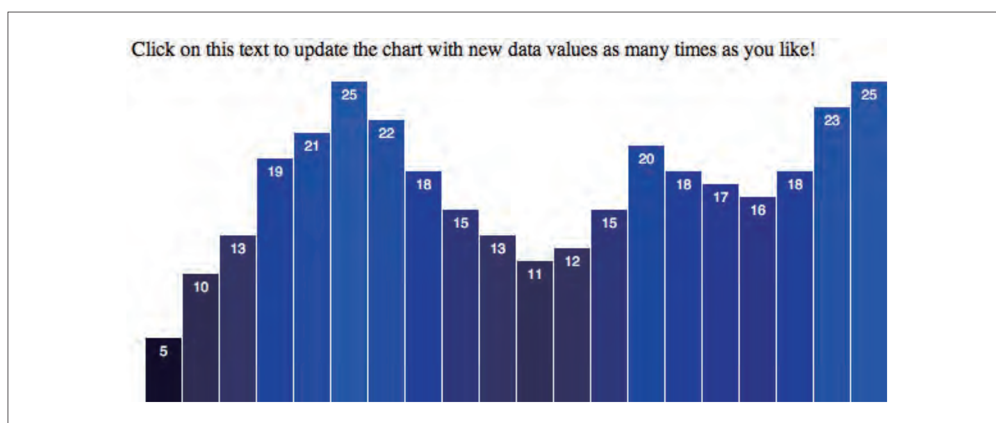


图 9-7：刚开始的图表

每次单击上面的段落，代码都会做如下这些事：

- 生成新随机数；
- 把新值绑定到已有元素；
- 把元素过渡到与新值对应的新位置、新高度、新颜色（参见图 9-8）。

太酷了！如果你觉得这不是很酷，或者一点也不酷，请关掉电脑，出去散散步，让脑子清醒一下，恢复一下对酷的感受力。如果你也觉得这酷得简直都不行了，好好，继续往下看。

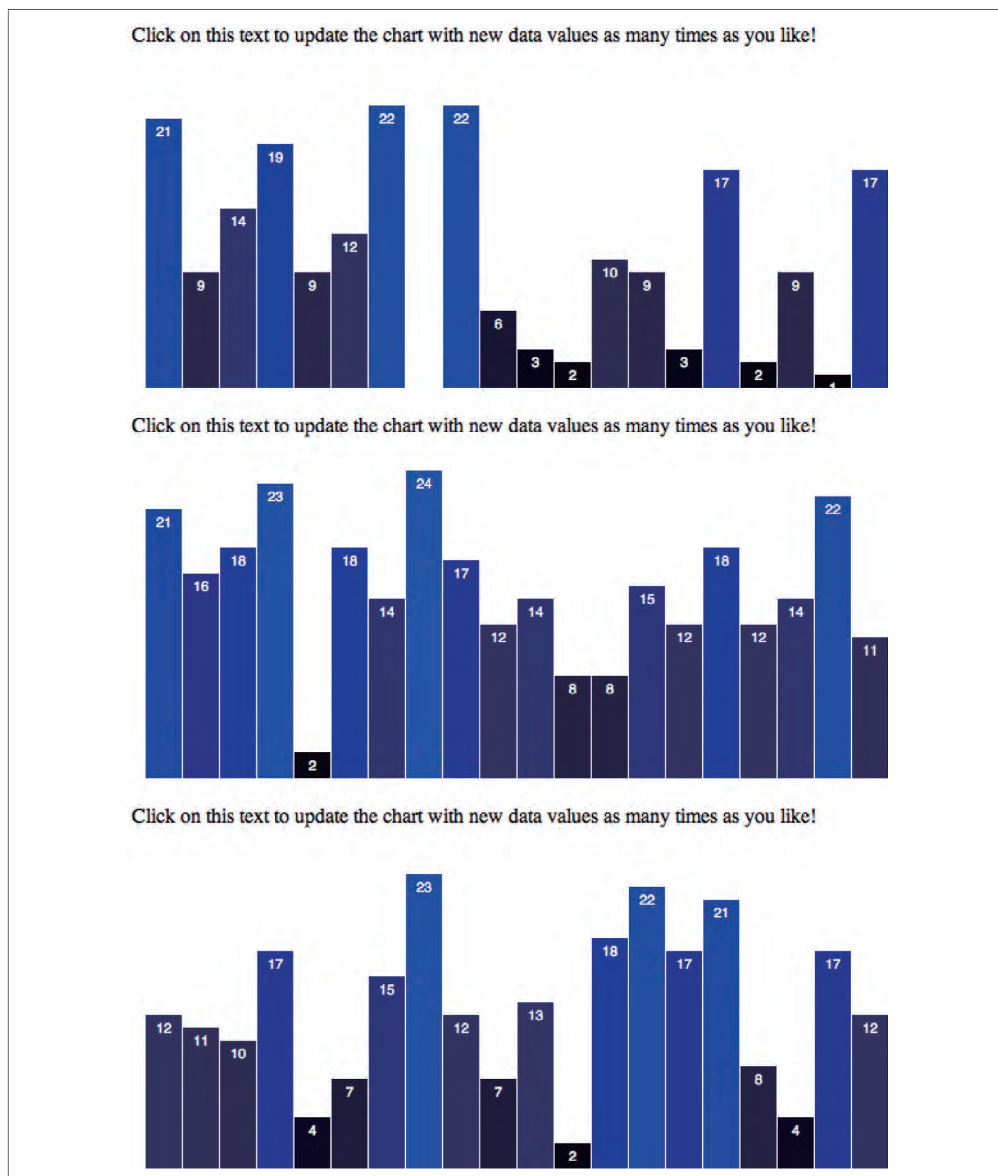


图 9-8：使用随机数据生成的图表每次都不同

9.3.5　更新比例尺

独具慧眼的读者可能会对前面的一行代码提出质疑：

```
var newNumber = Math.floor(Math.random() * 25);
```

为什么是 25？在编程领域，25 是一个幻数。我知道，你第一次听说，但幻数的问题就在于，虽然听起来很好玩，却谁都说不出来它为什么存在（“幻数”嘛，也难怪）。实际上，这里用 maxValue 代替 25 可能更有意义：

```
var newNumber = Math.floor(Math.random() * maxValue);
```

幻数不见了，但我心里知道这里生成的 newNumber 最大不会超过 25。作为一个约定，大家尽量不要使用幻数，还是把它保存在一个变量里，给变量起个有意义的名字更靠谱。起个什么名字有意义呢，maxValue 可以，numberOfTimesWatchedTheMovieTopSecret 当然也可以（这个变量的意思是“看电影《笑破铁幕》的次数”）。

更重要的是，我知道原来之所以选定 25，是因为更大的值超出了图表比例尺的输出范围，会导致条形被切掉。图 9-9 展示了把 25 换成 50 之后的后果。

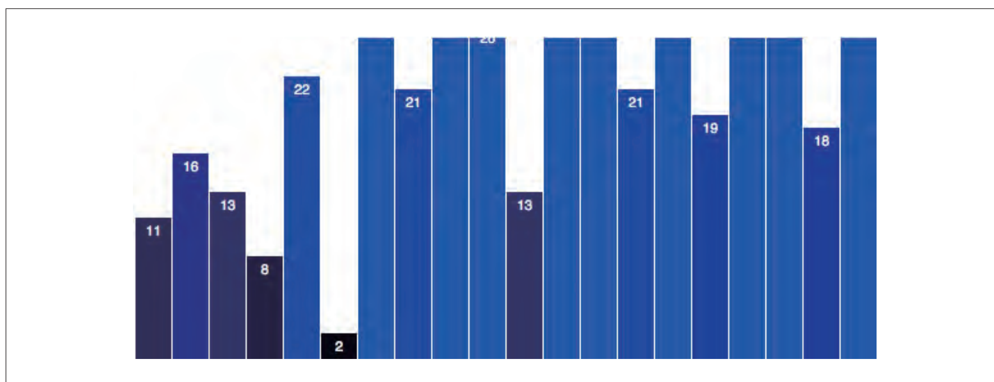


图 9-9：太高啦！搞错幻数了

问题的根源并不在于选错了幻数，而在于应该与数据集同步更新比例尺。每次插入新数据，都应该校准一下比例尺，确保生成的条形图不会过高或过低。

更新比例尺简单。还记得下面创建 yScale 的代码吗：

```
var yScale = d3.scale.linear()
                .domain([0, d3.max(dataset)])
                .range([0, h]);
```

范围不变（图表的可见大小没变），只要在生成新数据集之后更新比例尺的值域就好了：

```
// 更新比例尺的值域
yScale.domain([0, d3.max(dataset)]);
```

这样就把输入值域的最大值设置成了数据集中最大的数值。后面更新所有条形和标

签的代码本来就引用了 `yScale` 来计算各自的位置，因此就不用再修改其他代码了。

看一看 18_dynamic_scale.html 吧，我已经把幻数 25 替换成了 `maxValue`，而这个变量现在被设置为 100。此时单击段落更新数据，就能得到 0~100 之间的随机数。而且，如果数据集中最大的值是 100，那 `yScale` 的值域也会扩展到 100，如图 9-10 所示。

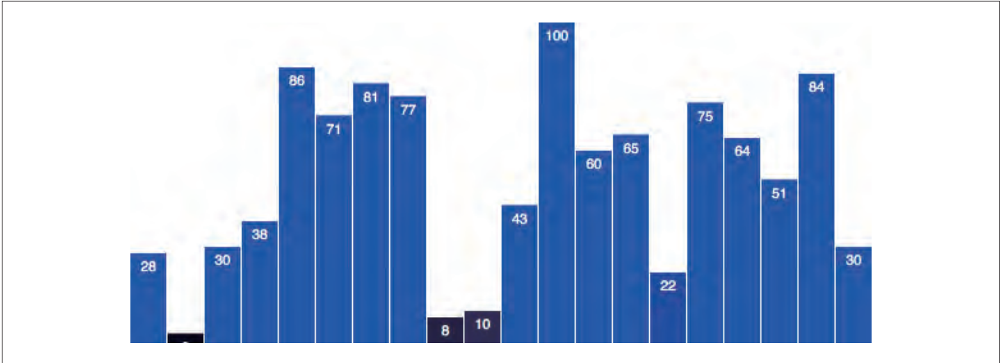


图 9-10：根据随机数据生成的条形图，但 *y* 轴比例能够自动适应了

由于数据是随机生成的，最大值不一定总是 100，所以在图 9-11 中能看到的最大数是 85。

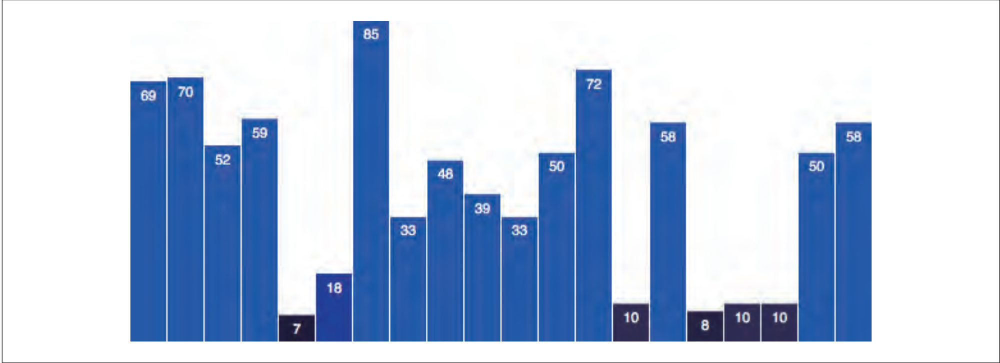


图 9-11：数据有了变化，因此比例尺也跟着变化了

要注意的是，图 9-10 中值为 100 的条形与图 9-11 中值为 85 的条形高度相同。因为数据变了，比例尺输入值域变了，而可见的输出范围则没变。

9.3.6 更新数轴

这个条形图没有数轴，但上一章那个散点图有啊（参见图 9-12）。把那个例子拿过

来，改一改，就是 19_axes_static.html。

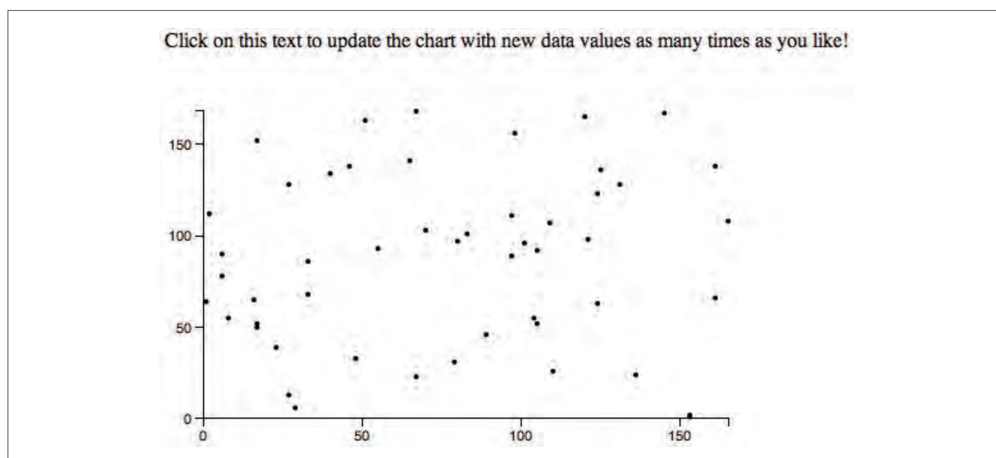


图 9-12：更新后的散点图，支持数据更新和动态比例尺

总结一下对这个散点图所做的改动。

- 单击上方的文本可以生成新数据并更新图表。
- 更新数据后，使用了动画过渡。
- 去掉了交错延迟效果，将所有过渡的持续时间都设定为 1 秒（1000 毫秒）。
- 同时也更新 *x* 和 *y* 轴的比例尺。
- 圆形现在有了固定的半径。

现在单击文本，看一看这些小圆点四处穿梭的效果吧。太可爱了！虽然我希望这些圆点可以表示有意义的信息，不过随机数据倒也挺有趣啊。

但图中的数轴没有变。不过，要更新数轴也很简单。

首先，给 *x* 轴和 *y* 轴分别添加类名 x 和 y，以便后面选择它们：

```
// 创建 x 轴
svg.append("g")
    .attr("class", "x axis")    // <-- 在这里添加了 x 类
    .attr("transform", "translate(0," + (h - padding) + ")")
    .call(xAxis);

//Create y-axis
svg.append("g")
    .attr("class", "y axis")    // <-- 在这里添加了 y 类
    .attr("transform", "translate(" + padding + ",0)")
    .call(yAxis);
```

然后，找到响应单击的匿名函数，在里面添加如下代码：

```
// 更新 x 轴
svg.select(".x.axis")
    .transition()
    .duration(1000)
    .call(xAxis);

// 更新 y 轴
svg.select(".y.axis")
    .transition()
    .duration(1000)
    .call(yAxis);
```

对每个数轴，我们分别做了以下几件事。

- 选择当前数轴。
- 初始化一个过渡。
- 设定过渡的持续时间。
- 调用适当的数轴生成器。

记住，每个数轴生成器已经引用了相应的比例尺（`xScale` 或 `yScale`）。由于这些比例尺已经更新过了，所以数轴生成器可以计算出新刻度的位置。

打开 20_axes_dynamic.html 试试吧。

同样，`transition()` 替我们完成了所有插帧操作。看看那些淡入淡出的刻度线，都是不费吹灰之力就能实现的。

9.3.7 在过渡开始和结束时执行操作

有时候，我们需要在过渡开始和结束的时候执行一些操作。为此，可以使用 `each()` 方法，它能对选中的每个元素执行任意代码。

`each()` 接收两个参数：

- `"start"` 或 `"end"`；
- 匿名函数，在过渡开始或结束时执行。

例如，在下面更新圆形的代码中，我们添加了两条 `each()` 语句：

```
// 更新所有圆形
svg.selectAll("circle")
   .data(dataset)
   .transition()
   .duration(1000)
   .each("start", function() {      // <-- 在过渡开始时执行
       d3.select(this)
         .attr("fill", "magenta")
```

```
            .attr("r", 3);
    })
    .attr("cx", function(d) {
         return xScale(d[0]);
    })
    .attr("cy", function(d) {
         return yScale(d[1]);
    })
    .each("end", function() {          // <-- 在过渡结束时执行
        d3.select(this)
          .attr("fill", "black")
          .attr("r", 2);
    });
```

可以看看 21_each.html 中的效果。

单击触发过渡动画后，立刻可以看到每个圆形都变成了粉红色（.attr("fill", "magenta")），而半径也变大了（.attr("r", 3)）。然后过渡效果一如往常。动画最后，填充和半径都恢复到原始状态。

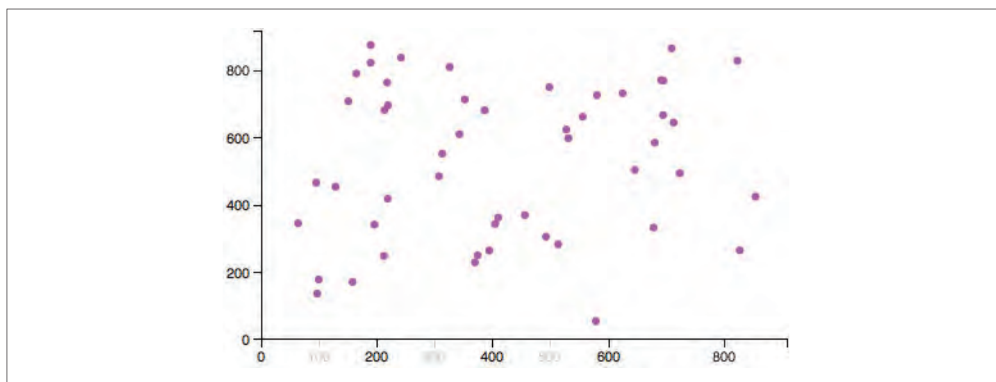


图 9-13：过渡期间的粉红圆形

这里需要注意传给 each() 的匿名函数。在这个匿名函数的上下文里，this 引用着“当前元素”。这样非常方便，通过它在函数里就可以随时取得当前选择的元素并修改之，如下所示：

```
    .each("start", function() {
        d3.select(this)                  // 选择 'this'，即当前元素
          .attr("fill", "magenta")     // 把 'this' 的填充设置为粉红
          .attr("r", 3);                // 把 'this' 的半径设置为 3
    })
```

1. 谨慎开局

你或许会忍不住在这里再加一个过渡动画，让圆形从黑色平滑过渡到粉红色。住手，别那么干！或者，试试吧，这样会造成整个过渡中断：

```
.each("start", function() {
    d3.select(this)
      .transition()               // 新过渡
      .duration(250)              // 新持续时间
      .attr("fill", "magenta")
      .attr("r", 3);
})
```

如果你这样干了——建议你这么干一次看看，你会发现圆形确实会过渡为粉红色，但它们的位置却不会再改变了。原因是，在默认情况下，任何元素在任意时刻都只能有一个过渡效果。新过渡效果会打断并覆盖原来的过渡效果。

这不是个设计缺陷吗？不，D3 这样设计是有用意的。假设页面上有几个不一样的按钮，每一个都会触发不同的数据视图，如果用户连续地把它们都单击一遍会怎么样？难道你不认为较早的过渡应该中止，好让后来选择的视图马上过渡进来吗？

如果读者熟悉 jQuery，会在这里发现不同之处。默认情况下，jQuery 会把所有过渡效果排成队列，然后一个接一个地执行它们。换句话说，执行新过渡不会自动中断原有过渡。这种设计有时候会导致令人讨厌的界面行为，比如鼠标放到菜单上再离开后，菜单并不会马上就淡出，而是必须完全淡入之后再淡出。

在这里，代码之所以“中断”是因为第一个（空间）过渡一开始，`each("start", ...)` 就在每个元素上执行。而在 `each()` 里头，又开始了第二个（颜色）过渡，这个过渡会覆盖第一个过渡，因而圆形永远不会再跑到它们的目标位置上了（尽管就那么呆在那儿看起来也不错）。

由于过渡中存在的这个问题，一定要记住只能在 `each("start", ...)` 里面执行立即变换，而不能再添加任何过渡效果。

2. 优雅收场

不过，`each("end", ...)` 则支持过渡。在执行 `each("end", ...)` 的时候，主过渡已经结束了，因此再执行新过渡不会产生任何副作用。

参见 22_each_combo_transition.html，在里面的第一个 `each()` 语句中，我把粉色的圆形半径增大到了 7。在第个 `each()` 语句中，我又添加了两行设置过渡和持续时间的代码：

```
.each("end", function() {
    d3.select(this)
      .transition()              // <-- 新代码!
      .duration(1000)            // <-- 新代码!
      .attr("fill", "black")
      .attr("r", 2);
});
```

看看这个过渡怎么样？太酷了！注意事件发生的顺序：

- 你单击 p 元素的文本；
- 圆形立即变成粉红并增大；
- 圆形过渡到新位置；
- 圆形过渡到原来的颜色和大小。

另外，试试一连串多单击几次 p 元素。快，别多想，尽可能快地多单击几次。你会发现每次单击都会中断圆形的过渡。（抱歉，伙计！）每个新过渡都会覆盖旧过渡。除非你停止单击，给它们时间过渡，否则圆形永远不会跑到它们的最终位置并过渡为黑色。

跟这个功能差不一样酷的是，还有一种方法能够把多个过渡安排在一起，一个接一个地执行。怎么做？只要把多个过渡连缀起来就行。（这是 3.0 版中的新功能，老版不支持。）比如，以前在 `each("end", ...)` 中必须重新选择元素，然后应用过渡，而现在只要在整个方法链最后追加一个过渡即可：

```
svg.selectAll("circle")
    .data(dataset)
    .transition()    // <-- #1 过渡
    .duration(1000)
    .each("start", function() {
            d3.select(this)
              .attr("fill", "magenta")
              .attr("r", 7);
    })
    .attr("cx", function(d) {
                return xScale(d[0]);
    })
    .attr("cy", function(d) {
                return yScale(d[1]);
    })
    .transition()    // <-- #2 过渡
    .duration(1000)
    .attr("fill", "black")
    .attr("r", 2);
```

试试包含这些代码的 23_each_combo_transition_chained.html，你会发现结果完全一样。

在需要顺序执行多个过渡时，建议你使用这种连缀方法。然后，对于恰好需要在过渡开始和结束时立即执行的（非过渡性）变化，使用 `each()` 方法。可以想见，通过连缀多个独立的动态变化的过渡，然后在这些过渡上再应用 `each("start", ...)` 和 `each("end", ...)`，能够创造出相当复杂的效果。

3. 无过渡的`each()`

还可以在过渡的外部使用 `each()`，作用就是针对每个元素执行任意代码。在过渡外部，可以忽略 `"start"` 或 `"end"` 参数，只要传入匿名函数即可。

虽然我们不会再举例子，但你可以在函数定义中包含对 `d` 和 `i` 的引用，D3 会帮你处理这些值：

```
...
.each(function(d, i) {
    // 对 d 和 i 进行处理
});
```

4. 在剪切路径中包含可见元素

不知道大家注意没注意一个细节，在这些过渡中，那些 *x* 和 *y* 值较低的圆形会超出图表区域的边界，与轴线重叠在一起，如图 9-14 所示。

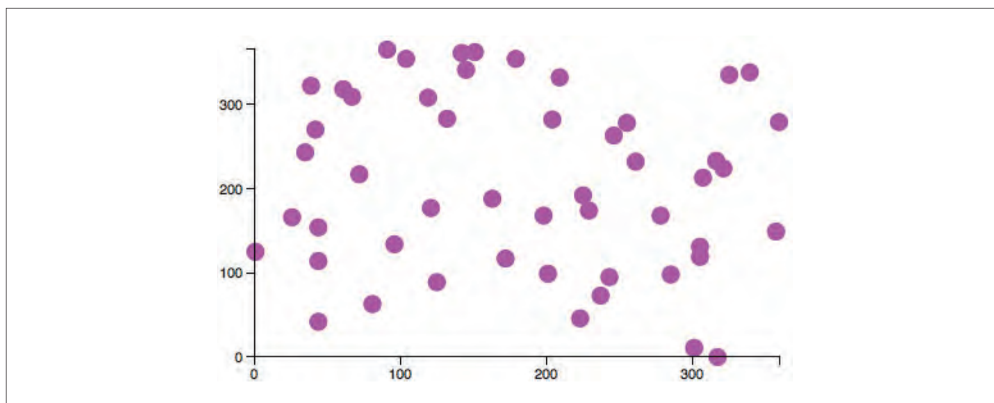


图 9-14：圆形超出了图表区域

好在 SVG 支持剪切路径（clipping：path），也就是 Photoshop 或 Illustrator 中的蒙版。剪切路径就是一个 SVG 元素，可以包含可见的元素，并与这个可见元素一起构成可以应用到其他元素的剪切路径或蒙版。在把蒙版应用到某个元素时，只有落在该蒙版图形内部的像素才会显示。

与 `g` 元素相似，`clipPath` 本身也不可见，但它可以包含可见的元素（这些元素用作蒙版）。比如，下面就是一个简单的剪切路径：

```
<clipPath id="chart-area">
    <rect x="30" y="30" width="410" height="240"></rect>
</clipPath>
```

注意外面的 `clipPath` 元素有一个 ID，值为 `chart-area`。后面会通过这个 ID 来

引用它。这个剪切路径内部有一个矩形，将被用作蒙版。

使用剪切路径的步骤如下：

1. 定义 clipPath 并给它一个 ID；
2. 在这个 clipPath 中放一个可见的元素（通常是一个 rect，但也可以包含 circle 和其他可见元素）；
3. 在需要使用蒙版的元素上添加一个对 clipPath 的引用。

仍以前面的散点图为例，下面我们通过一些新代码来定义剪切路径（完成第 1 步和第 2 步）：

```
// 定义剪切路径
svg.append("clipPath")                    // 创建新的 clipPath 元素
    .attr("id", "chart-area")             // 为它指定 ID
    .append("rect")                       // 在 clipPath 中，创建并添加新的 rect
                                             元素
    .attr("x", padding)                   // 设置 rect 的位置和大小……
    .attr("y", padding)
    .attr("width", w - padding * 3)
    .attr("height", h - padding * 2);
```

我们希望把这个蒙版应用给所有圆形，可以分别为每个圆形都添加一个对该 clipPath 的引用。但把所有圆形都放到一个分组 g 中，然后给这个分组添加引用，会让代码更清晰也更简单（完成第 3 步）：

因此，我们要修改以下代码：

```
// 创建圆形
svg.selectAll("circle")
   .data(dataset)
   .enter()
   .append("circle")
   ...
```

创建新的 g 元素，给它任意指定一个 ID，最后把对 ID 为 chart-area 的剪切路径指定给它：

```
// 创建圆形
svg.append("g")                                  // 创建新的 g 元素
   .attr("id", "circles")                        // 指定它的 ID 为 circles
   .attr("clip-path", "url(#chart-area)")        // 添加对 clipPath 的引用
   .selectAll("circle")                          // 剩下的代码跟以前一样……
   .data(dataset)
   .enter()
   .append("circle")
   ...
```

注意在指定剪切路径时使用的属性名叫 `clip-path`，而这个元素的名称是 `clipPath`。啊！我知道，我也会快疯了。

还是先看看示例页面 24_clip-path.html 吧，在 Web 检查器中可以看到新创建的矩形，如图 9-15 所示。

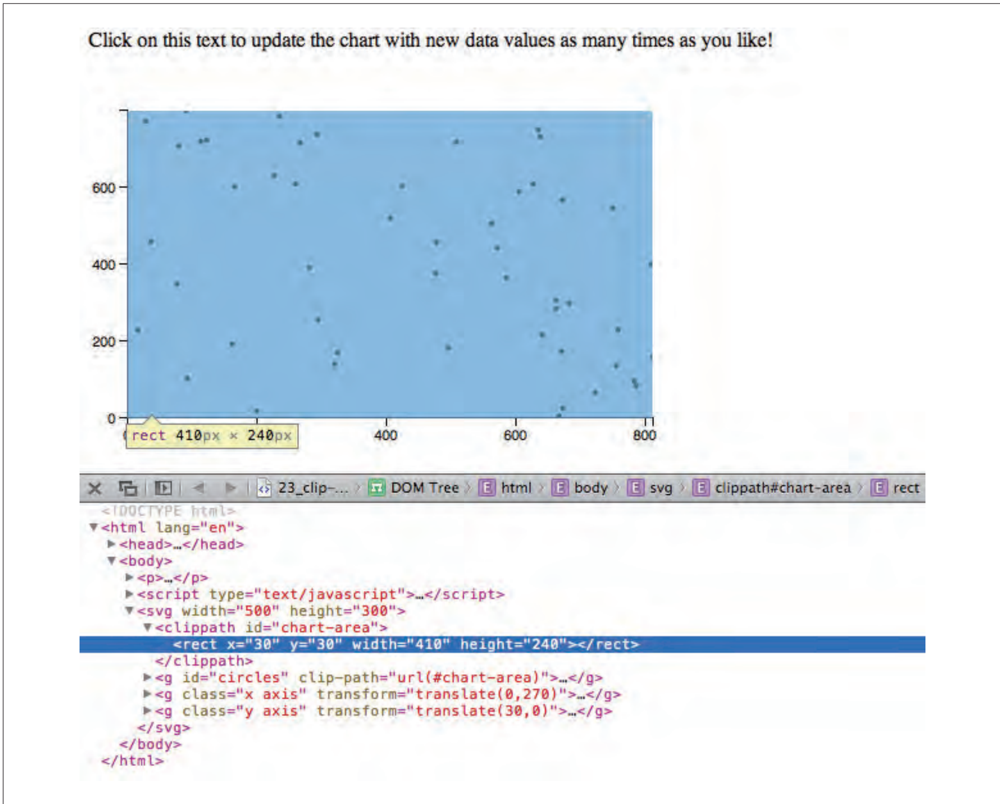


图 9-15：位于剪切路径中的矩形的大小

由于剪切路径本身不可见（只用于遮挡其他元素），因此只能通过 Web 检查器来高亮它。在 Web 检查器中选择这个 `clipPath` 后，就会看到蓝色的路径的轮廓、大小。在图 9-15 中，可以看到这个 `clipPath` 的位置和大小都合适。

另外也要注意，所有圆形都已经被组织到一个 `g` 元素中了。这个 `g` 元素的 `clip-path` 属性指向了新创建的剪切路径，语法有点不太常见：`url(#chart-area)`。但这都是 SVG 标准规定的。

最终结果就是，当圆形跑到图表区域的边界时，超出边界的部分会被剪切掉。注意最上方和最右边的那些圆形。

剪切效果在过渡期间比较明显，参见图 9-16。

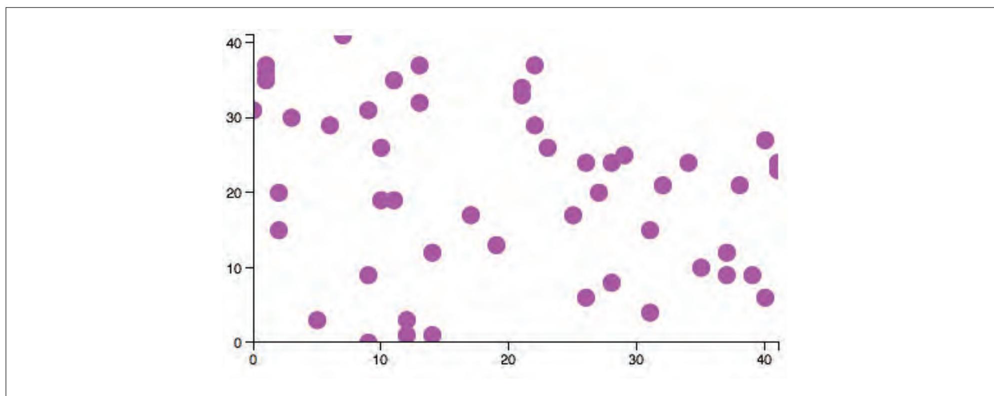


图 9-16：圆形都包含在图表区域中了

好啦！散点们都不会再跑到图表区域外面去了。

9.4 其他数据更新方式

迄今为止，只要更新数据，我们采用的都是“整批整包”的方式：改变数据集数组中的值，然后重新绑定修改后的值，覆盖原始值对 DOM 元素的绑定。

这种方式非常适合所有值都会改变，而且数据集长度（即数据值的数量）不变的情形。可是我们知道，现实中的数据可没那么简单。这就对代码的灵活性提出了更高要求，比如只更新一两个值，或者支持增加值和减少值，等等。而这些对 D3 来说，都是手到擒来的事。

9.4.1 添加值（和元素）

再拿我们可爱的条形图作例子，假设某些用户交互（单击鼠标）会触发向数据集中添加新值。换句话说，数据集的长度每次会加 1。

好，生成随机数并将其添加到数组太简单了：

```
// 向数据集中添加一个新值
var maxValue = 25;
var newNumber = Math.floor(Math.random() * maxValue);
dataset.push(newNumber);
```

要给新增的条形留出空间来，必须对 x 轴的比例尺进行校正。而这也不过是更新其输入值域，以反映新数据集长度的问题：

```
xScale.domain(d3.range(dataset.length));
```

很简单。注意，接下来准备费些脑筋吧，我们要深入讨论 D3 选择元素的问题了。

1. 选择

到现在为止，我们已经习惯使用 `select()` 和 `selectAll()` 取得和返回 DOM 元素。而且，也知道把这些方法连缀起来，就可以从已经选择的元素中进一步选择元素，比如：

```
d3.select("body").selectAll("p");    // 返回 body 中的所有 p 元素
```

在需要保存这些选择方法返回的结果时，我们知道最终结果（也就是方法链中最后一个选择符匹配的元素）是一个对选中元素的引用，可以保存一个变量中：

```
var paragraphs = d3.select("body").selectAll("p");
```

这样，`paragraphs` 就包含了从 DOM 中选择的所有 `p` 元素，无论 `p` 元素位于 DOM 中的哪个层次。

比较费解的地方是 `data()` 方法也会返回一个元素集（selection）。特别地，`data()` 返回的是对刚刚绑定了数据的所有元素的引用，可以称之为更新元素集。

对条形图来说，这就意味着可以选择所有条形，然后把新数据一次重新绑定到它们：

```
// 选择……
var bars = svg.selectAll("rect")
    .data(dataset);
```

这样，`bars` 中就保存了更新元素集。

2. 加入

在修改数据值而整个数据集的长度不变时，不用去管更新元素集。因为只要重新绑定数据，并将新值反映到新属性值上即可。

可现在新增了一个值。因此 `dataset.length` 从原来的 20 变成了 21。怎么处理这个新值，尤其是怎么为它绘制一个新矩形呢？希望大家耐心一点，我会给大家讲清楚。

更新元素集的天才之处在于，它在内部保存着对那些加入和退出的子元素集的引用。

加入元素就是那些新增的元素。而随时准备欢迎这样的元素是一种有涵养的表现。

只要绑定的数据值比对应的 DOM 元素多，则加入元素集中就会包含那些还不存在的元素。我们已经知道怎么访问这个加入元素了：在绑定新数据后，调用 `enter()`

方法。这在我们当初创建条形图的时候就用到过。还记得以下代码吧：

```
svg.selectAll("rect") // 选择所有矩形（尽管还不存在矩形）
    .data(dataset)     // 绑定数据到元素集，返回更新元素集
    .enter()           // 提取加入元素，比如 20 个占位元素
    .append("rect")    // 在每个占位元素中分别创建一个 rect
    ...
```

像这种 selectAll() → data() → enter() → append() 连缀的调用形式，我们已经见过很多次了。但每次都是在页面加载时一次性创建很多元素。而现在数据集中多了一个值，同样可以使用 enter() 来处理与这个新值对应的 DOM 元素，不会影响已有的所有矩形。在前面的选择代码之后，再添加如下代码：

```
// 加入……
bars.enter()
     .append("rect")
     .attr("x", w)
     .attr("y", function(d) {
         return h - yScale(d);
     })
     .attr("width", xScale.rangeBand())
     .attr("height", function(d) {
         return yScale(d);
     })
     .attr("fill", function(d) {
         return "rgb(0, 0, " + (d * 10) + ")";
     });
```

别忘了，bars 包含着更新元素集，因此 bars.enter() 会从中提取出新加入的元素。在这里，新加入的元素就是一个对 DOM 元素的引用。然后，调用 append() 创建这个新 rect，再后面的 attr() 语句一如往常，只有下面这一行不一样：

```
.attr("x", w)
```

没错，这行代码设定了新条形的水平位置，让它恰好位于 SVG 区域的最右边。我是希望这个新条形一开始位于视野之外，这样才好使用平滑的过渡效果，优雅地把它请进来。

3. 更新

创建了新 rect，剩下要做的就是更新所有矩形的可见属性了。同样，这里的 bars 保存着完整的更新元素集（也包括加入元素）：

```
// 更新……
bars.transition()
     .duration(500)
     .attr("x", function(d, i) {
         return xScale(i);
```

```
    })
    .attr("y", function(d) {
        return h - yScale(d);
    })
    .attr("width", xScale.rangeBand())
    .attr("height", function(d) {
        return yScale(d);
    });
```

以上代码会让所有条形都过渡到它们的新 `x`、`y`、`width` 和 `height` 值。不信？看看示例页面 25_adding_values.html。图 9-17 展示了开始时的条形图。

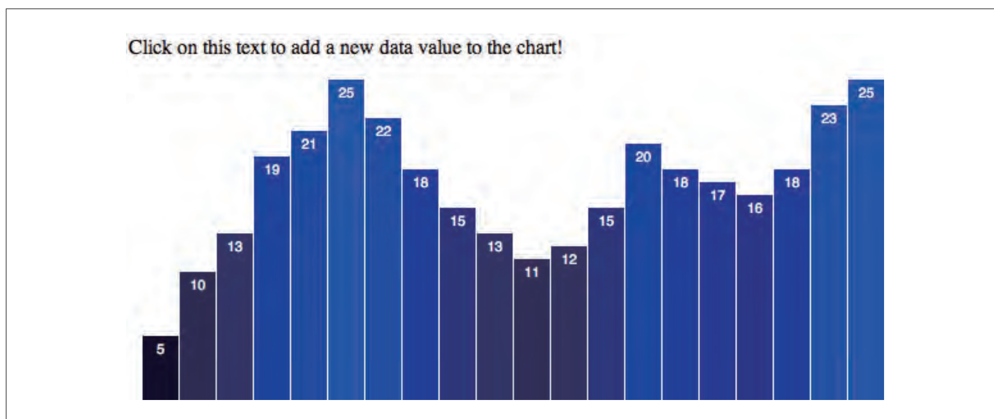


图 9-17：初始状态的条形图

图 9-18 展示了单击文本之后的条形图。注意右边多出来的新条形。

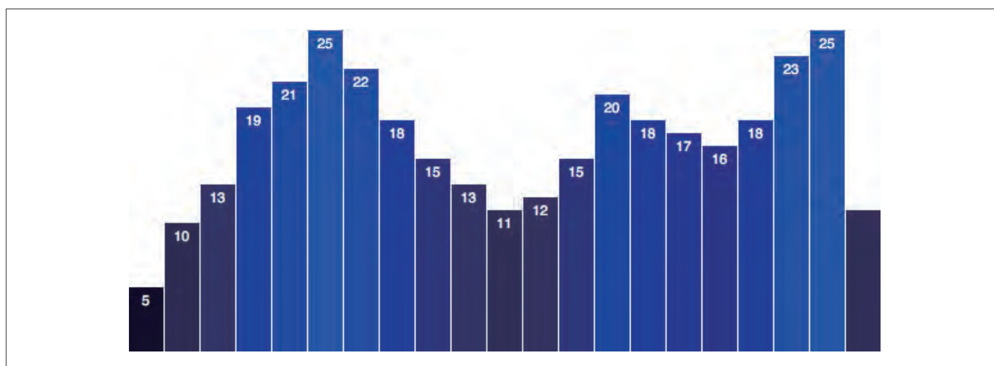


图 9-18：单击之后的条形图

两次单击后，就得到了图 9-19 所示的条形图。图 9-20 展示的是三次单击后的效果。

再多单击几次，可以看到图 9-21 所示的条形图。

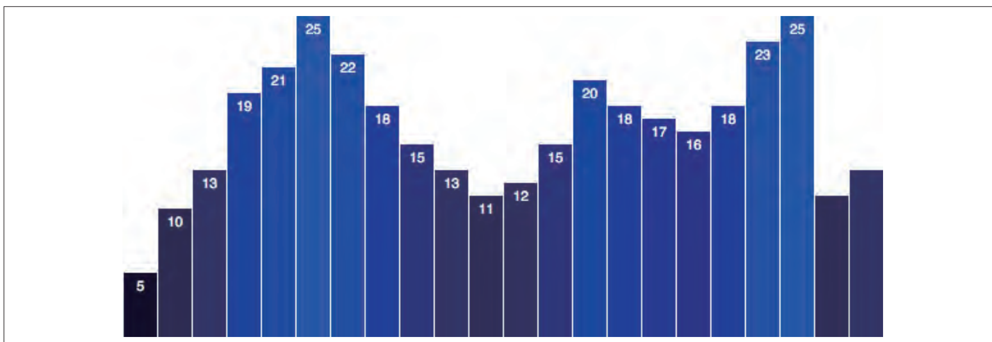


图 9-19：两次单击之后的条形图

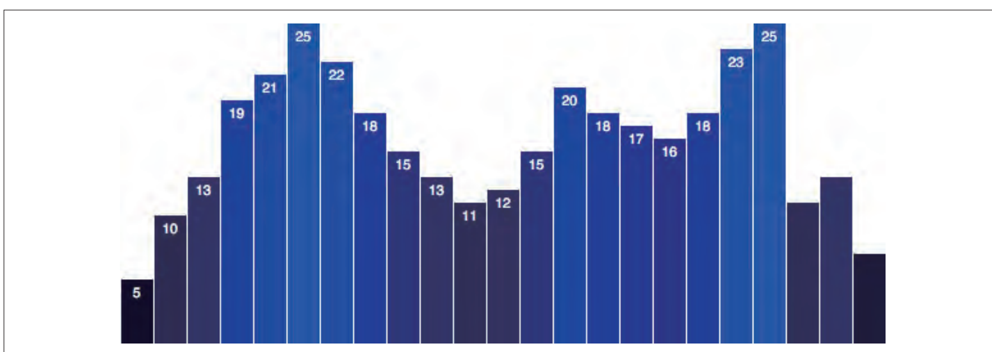


图 9-20：三次单击之后的条形图

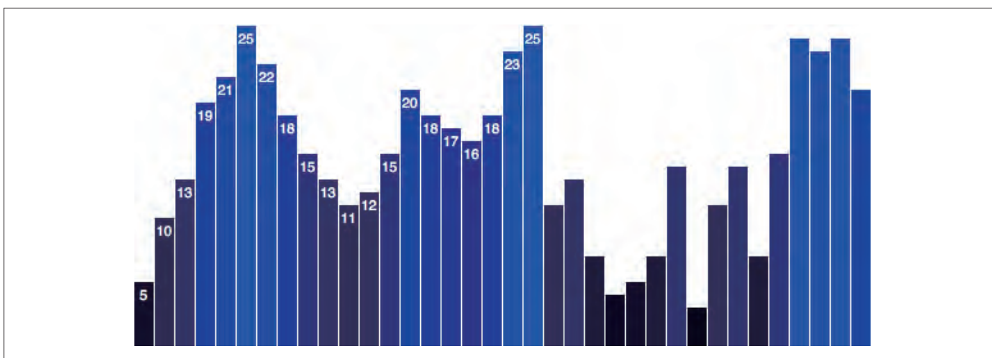


图 9-21：多次单击之后的条形图

不仅每次单击都会生成新条形，新条形能够自动调整大小，自动定位，而且每次单击之后，其他所有条形也会重新调整大小，重新排列。

现在还需要做的是为新条形创建值标签，并将它们过渡到位。这个嘛，就留给大家当练习了。

9.4.2 删除值（和元素）

删除元素更简单一些。

如果 DOM 元素比数据值还多，那么退出元素集中会包含那些没有对应数据的元素的引用。想必大家已经猜到了，要访问退出元素集，可以使用 `exit()` 方法。

首先，我们改一改触发文本，说明要删除值：

```
<p>Click on this text to remove a data value from the chart!</p>
```

然后，把每次单击时生成新随机值并添加到数据集，改为使用 `shift()` 方法从数组中删除第一个元素：

```
// 从数据集中删除一个值
dataset.shift();
```

1. 退出

退出元素是指那些该删除的元素。对这些元素，我们应该礼貌有加，祝福它们旅途愉快。

好，先取得退出元素集，然后把它们过渡到右边，最后，删除它们：

```
// 退出……
bars.exit()
    .transition()
    .duration(500)
    .attr("x", w)
    .remove();
```

这里的 `remove()` 是一特殊的过渡方法，它会在过渡完成后从 DOM 中永远地删除元素。（抱歉，再也找不回来了。）

2. 温和退出

从视觉角度说，先执行过渡而不是简单地调用 `remove()` 把元素删除是个不错的做法。在这里，我们是先把条形移动到右边，但把它们的不透明度过渡为 0，或者应用其他过渡效果也很容易。

即便如此，假如你就希望尽快把元素删除，那就直接调用 `remove()`，去掉开头的过渡效果也没问题。

好啦，打开示例文件 26_removing_values.html 看一看吧，图 9-22 展示了初始状态的条形图。

单击一次文本，注意图 9-23 已经少了一个条形。

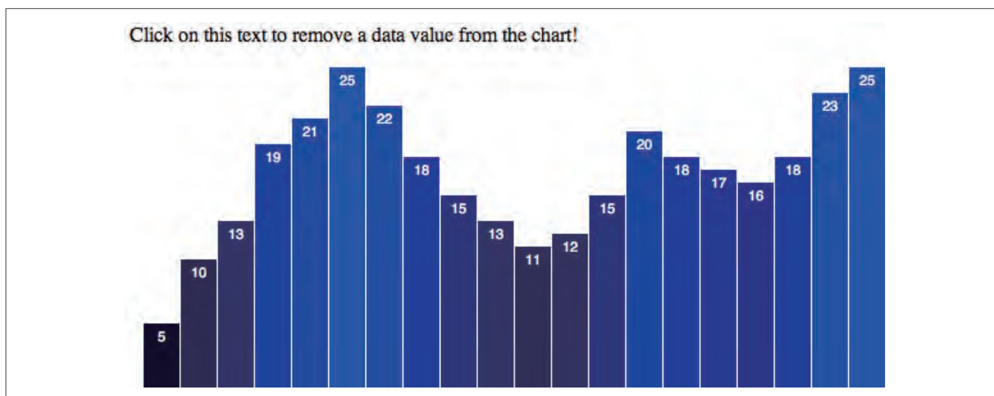


图 9-22：初始状态的条形图

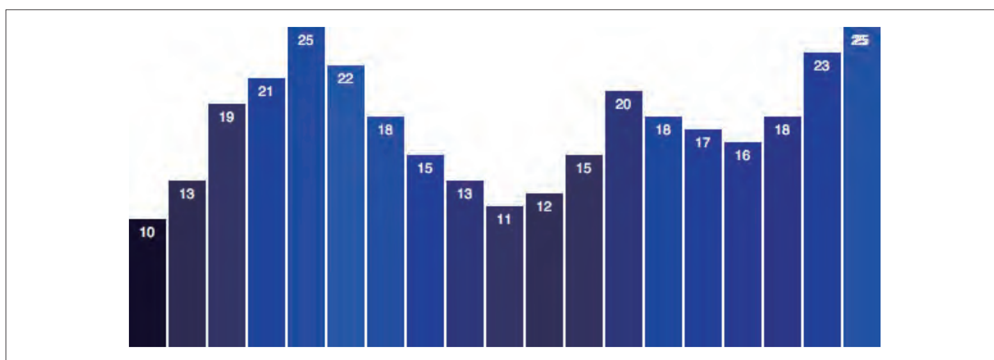


图 9-23：单击之后的条形图

单击两次之后，可以看到图 9-24 所示的效果。图 9-25 展示了单击三次后的效果。

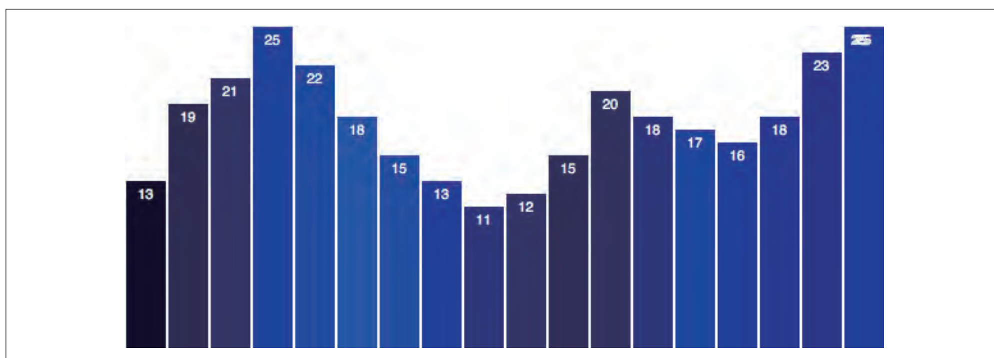


图 9-24：单击两次之后的条形图

多次单击之后，可以看到图 9-26 所示的结果。

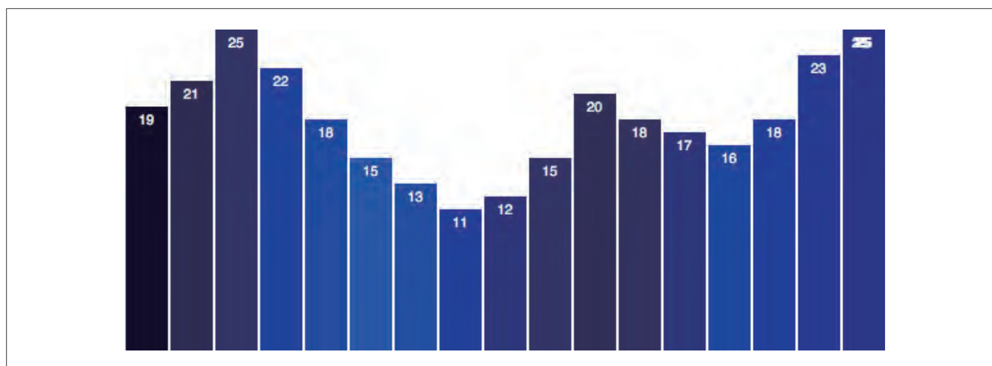


图 9-25：单击三次之后的条形图

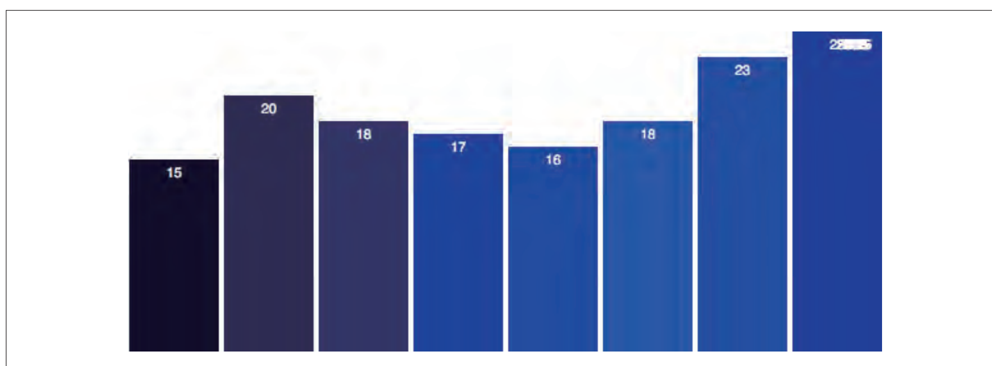


图 9-26：单击多次之后的条形图

每单击一次，都会有一个条形移动到右边，然后从 DOM 中销声匿迹。（可以在 Web 检查器中确认这一点。）

等等，好像哪里不对劲啊？首先，值标签并没有消失，每个被删除的条形都会在右上角留下一个标签，越来越乱。同样，删除值标签也作为一个练习留给大家吧。

更重要的是，虽然我们使用 `Array.shift()` 方法从数据集中删除的是第一个值，但却没有删除图中的第一个条形！是吗？是啊，你没注意到嘛，每次单击都是最右边的那个条形被删除。数据集更新没有问题（注意每个值消失的顺序分别是 5、10、13、19……），条形被重新赋予了新值，并没有“坚持”显示原来的值。换句话说，数据与条形的一致性（或目标一致性，object：constancy）出了问题，即原来标签为“5”的条形现在显示“10”了，依此类推。从直觉来讲，我们希望标签为“5”的条形自己从左边闪退，而其他条形则继续显示原来的值。

为什么，为什么，怎么会发生这种事?！别急，别急。这是有原因的，请大家听

我解释。维护数据与条形一致性的关键在哪儿？就在键上。（顺便告诉大家，Mike Bostock 关于对象值一致性写过一篇很中肯的文章，推荐大家读一读：http://bost.ocks.org/mike/constancy/。）

9.4.3 通过键联结数据

理解了更新、加入和退出元素集，接下来就可以深入探讨一下数据联结（data join）问题了。

在把数据绑定到 DOM 元素时（即调用 `data()` 时），就会发生数据联结。默认的联结是按照索引顺序，即第一个值绑定到元素集中第一个 DOM 元素，第二个值绑定到元素集中第二个 DOM 元素，依此类推。

如果数据值与 DOM 元素的顺序不一样呢？那就得告诉 D3 怎么实现值和元素间的联结或配对。好在，通过定义键函数（key：function），可以指定相应的规则。

这就解释了我们前面条形图的问题。在删除数据集中第一个值之后，我们又在既有元素上重新绑定了数据集。而数据集中的值是按照索引顺序联结的，因此第一个矩形（原来的值为 5），现在就会对应 10。而之前对应 10 的矩形，现在就会对应 13，依此类推。最终，还剩下一个 `rect` 元素没有对应的数据值，也就是最右边的那个条形。

通过定义键函数可以控制数据联结的具体方式，确保正确的数值绑定到正确的 `rect` 元素。

1. 准备数据

到目前为止，我们的数据集就是包含简单数值的数组。而为了使用键函数，每个值必须有对应的键。什么是“键”？就是值的唯一标识符，因为值本身可能会变，也可能会存在相同的值。（如果数据集里有两个值都是 3，那怎么区分它们呢？）

下面我们把数组改一改，改成对象的数组，让每个对象都包含一个键和一个实际的数据值：

```
var dataset = [ { key: 0, value: 5 },
                { key: 1, value: 10 },
                { key: 2, value: 13 },
                { key: 3, value: 19 },
                { key: 4, value: 21 },
                { key: 5, value: 25 },
                { key: 6, value: 22 },
                { key: 7, value: 18 },
                { key: 8, value: 15 },
                { key: 9, value: 13 },
                { key: 10, value: 11 },
                { key: 11, value: 12 },
```

```
                  { key: 12, value: 15 },
                  { key: 13, value: 20 },
                  { key: 14, value: 18 },
                  { key: 15, value: 17 },
                  { key: 16, value: 16 },
                  { key: 17, value: 18 },
                  { key: 18, value: 23 },
                  { key: 19, value: 25 } ];
```

还记得吗，方括号（[]）表示数组，花括号（{}）表示对象。

注意，这里的值其实并没有变，只是新增了很多键，而这些键也不过是列举了每个对象在数据集中的位置。（顺便澄清一下，这里的键名不一定是 key，也可以是 id、year 或 fruitType。这里使用 key 不过是为了简单。）

2. 更新所有引用

下一步并不好玩，但也不难。既然我们把数据都封装到了对象里，就不能再引用 d 了。（啊，再见了，过去的美好时光。）代码中任何要访问实际数据值的地方，都必须改为 d.value。在 D3 的方法中使用匿名函数时，d 永远是数组中当前位置的值。而现在，数组的每个位置上都是对象，类似 { key: 12, value: 15 } 。因此要取得 15，现在必须通过 d.value 了。

首先，必须得修改 yScale 的定义：

```
var yScale = d3.scale.linear()
                  .domain([0, d3.max(dataset, function(d) { return d.value;
})])
                  .range([0, h]);
```

第二行原来是简单的 d3.max(dataset)，但那只对简单的数组有效。现在我们使用的是对象数据，就必须包含一个访问器函数，让它告诉 d3.max() 怎么取得要比较的值。d3.max() 会遍历数组中所有的元素，而现在它知道不能再比较 d 了（因为 d 是一个对象，要比较对象可不容易）。

另外 ，还要修改响应单击的匿名函数中对 yScale 的另一处引用：

```
yScale.domain([0, d3.max(dataset, function(d) { return d.value; })]);
```

接下来，因为所有设置属性的地方都使用了 d，所以必须把它们都改为 d.value。比如这里：

```
...
.attr("y", function(d) {
    return h - yScale(d);         // <-- d
})
...
```

要改成这样：

```
...
.attr("y", function(d) {
    return h - yScale(d.value);  // <-- d.value！
})
...
```

3. 键函数

最后，我们再来定义键函数，以备把数据绑定到元素时使用：

```
var key = function(d) {
    return d.key;
};
```

注意到了吗，这个函数的参数也是 `d`，这是 D3 的惯例。当然，这个函数非常简单，只读取了传入的参数 `d` 的属性 `key` 的值。

现在，在四个绑定数据的地方都把这行代码：

```
.data(dataset)
```

改成这样：

```
.data(dataset, key)
```

对了，除了先定义键函数后引用，当然也可以直接把键函数写在调用 `data()` 的代码里，比如：

```
.data(dataset, function(d) {
    return d.key;
})
```

可这样一来，四个地方就得写四遍，有点太累赘了。因此，还是先定义键函数后引用更简洁。

这就行了！新数据联结完毕。

4. 退出过渡

最后还要解决一个问题，我们不是说过要让退出的条形从左边闪退，而不是从右边遁形嘛：

```
//退出……
bars.exit()
    .transition()
    .duration(500)
    .attr("x", -xScale.rangeBand())  // <-- 从左侧下台
    .remove();
```

太好了！看看包含所有新代码的 27_data_join_with_key.html。刚打开它时的界面应该如图 9-27 所示。

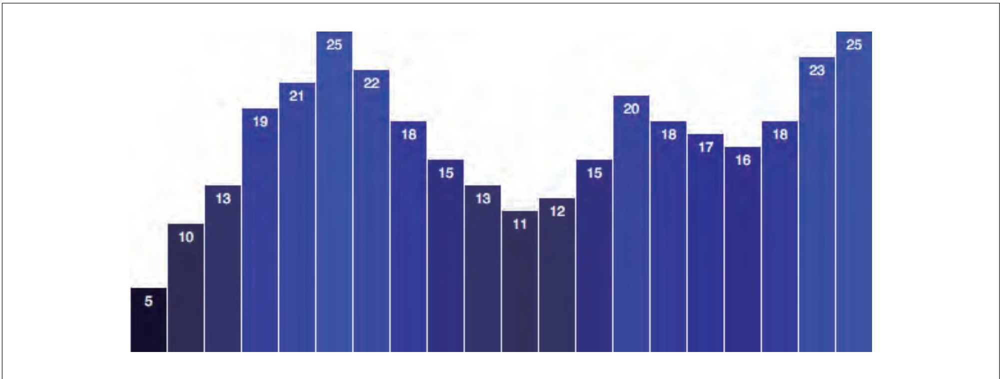


图 9-27：初始状态下的条形图

单击文本，则最左边的条形麻利地向左退出，而其他所有条形会重新调整到合适宽度，然后退出的条形从 DOM 中被删除。（同样，可以在 Web 检查器中看到 `rect` 元素一个一个地消失。）

图 9-28 展示了删除一个条形后的样子。

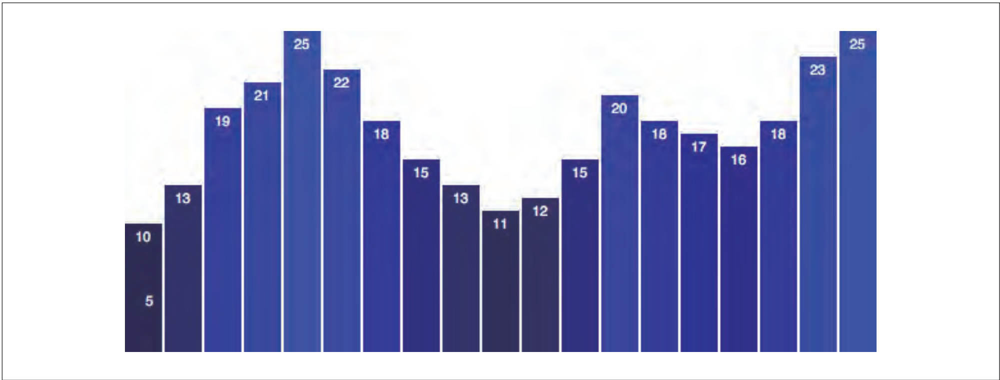


图 9-28：单击一次后的条形图

单击两次，看到的结果如图 9-29 所示。

图 9-30 显示了单击三次后的条形图，而更多次单击之后，可以看到图 9-31 所示的结果。

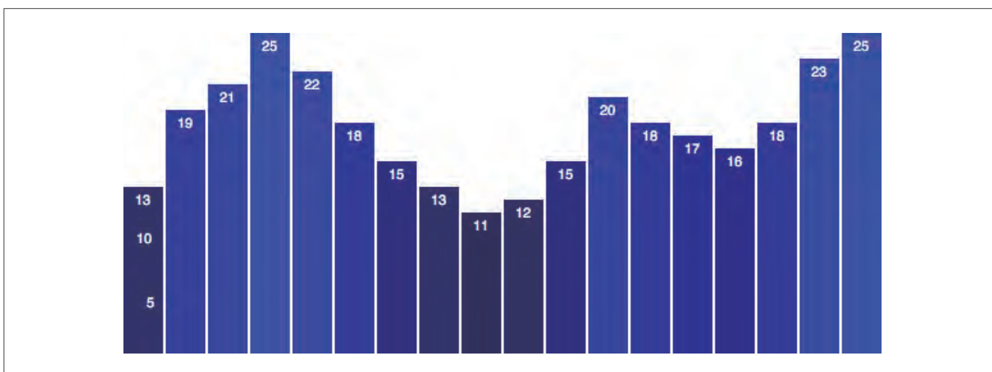


图 9-29：单击两次后的条形图

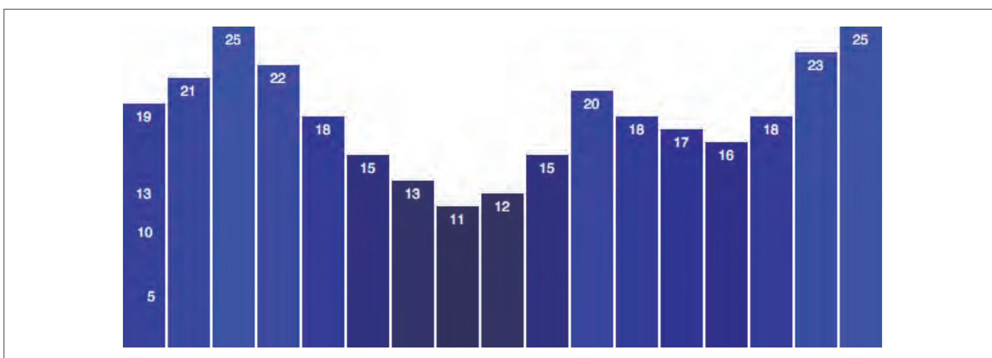


图 9-30：单击三次后条形图

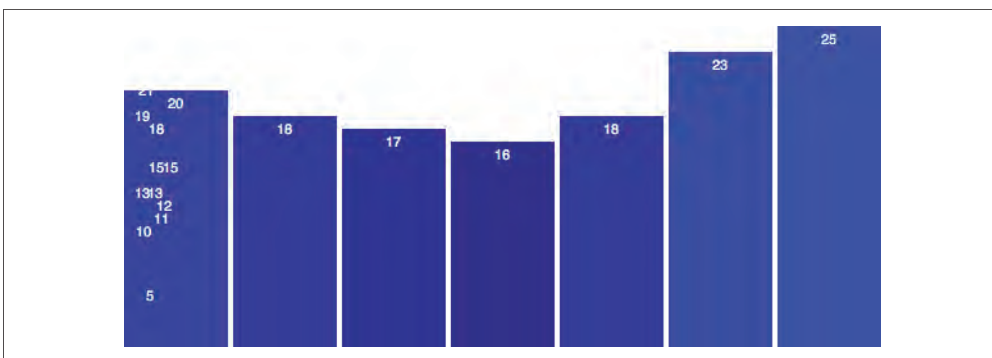


图 9-31：单击多次后的条形图

这一回比之前要好多了。唯一的问题就是标签没有从左侧退出，也没有从 DOM 中删除，因此图表左侧显得很乱。同样，这个问题还是交给你解决吧。把你新掌握的 D3 技术拿出来演练一番，把那些标签都清理干净！

9.4.4 添加和删除组合拳

现在似乎可以停手了，新掌握的技术让我们非常满足了。可是，为什么不能有始有终，再调整一下图表，让它同时支持添加和删除数据值呢？

这比你想象的简单得多。首先，需要两个不同的触发器，好让用户单击。为此，我们再增加一个段落，给两个段落各指定一个 ID，以便知道用户单击了哪一个：

```
<p id="add">Add a new data value</p>
<p id="remove">Remove a data value</p>
```

接下来，找到添加事件监听器的地方，把 `select()` 改成 `selectAll()`，这样才能选中多个 `p` 元素：

```
d3.selectAll("p")
    .on("click", function() { ...
```

既然一个函数被绑定到了两个段落，那就得添加一些逻辑判断，以告知函数根据用户单击的段落作出不同的处理。实现这种判断的方式非常多，我们只介绍最直观的一种。

没忘记吧，在匿名的监听器函数内部，`this` 引用被单击的元素（`p`），因此选择 `this` 再使用 `attr()` 就可以取得被单击元素的 ID：

```
d3.select(this).attr("id")
```

这行代码在用户单击 p#add 时会返回 `"add"`，在用户单击 p#remove 时会返回 `"remove"`。我们把这个值保存在变量中，根据它就可以控制 `if` 语句的行为：

```
//看看单击了哪个段落
var paragraphID = d3.select(this).attr("id");

//确定接下来该干什么
if (paragraphID == "add") {
    //Add a data value
    var maxValue = 25;
    var newNumber = Math.floor(Math.random() * maxValue);
    var lastKeyValue = dataset[dataset.length - 1].key;
    console.log(lastKeyValue);
    dataset.push({
        key: lastKeyValue + 1,
        value: newNumber
    });
} else {
    //删除一个值
    dataset.shift();
}
```

因此，如果单击的是 `p#add`，就计算一个新的随机值，然后查找数据集中最后一项的键值。把该键值加 1（以免与其他键冲突）作为新对象的键，把新随机值作为该对象的值，创建新对象。

其他地方不用再改了！我们写的这些加入 / 更新 / 退出代码已经足够灵活，可以处理添加和删除数据值的操作了。这正是代码之美。

试一试 28_adding_and_removing.html 吧。你可以随意单击来添加或删除数据值。当然，现实中的数据并不是这样生成的，但你想啊，其他事件也可以触发这些数据的更新。比如，用来自服务器的数据更新当前数据集，根本不用单击鼠标。

同样，再看看 29_dynamic_labels.html，大同小异，区别只在于我已经更新了代码，实现了标签的添加、过渡和删除。

9.4.5　简要回顾

这一章讲的内容太多啦！好吧，咱们简单回顾一下。

- `data()` 把数据绑定到元素，但也会返回*更新元素集*。
- 更新元素集可能包含加入和退出元素集，这两个元素集可以通过 `enter()` 和 `exit()` 方法得到。
- 在数据值比元素多的情况下，加入元素集会引用尚不存在的占位元素。
- 在元素比数据值多的情况下，退出元素集会引用没有对应数据的元素。
- 数据联结用于确定数据值怎么与元素匹配。
- 默认情况下，数据联结按照索引（也就是值在数据集中出现的顺序）进行。
- 要对数据联结施加更多控制，可以指定键函数。

最后一点，在条形图的示例中，我们使用了以下顺序：

1. 加入
2. 更新
3. 退出

尽管这个顺序可以使用，但不代表任何时候都只能是这个顺序。具体要看你的设计目标，可能需要先更新，然后加入新元素，最后退出旧元素。总之，完全要因时、因地制宜。记住一条，只要手里有更新元素集，可以随时取得加入和退出元素集。三者的顺序是灵活的，取决你的需要。

太不可思议了，你在成为 D3 专家的路上越走越远了。现在该来点真正有意思的东西了，那就是交互！

第10章

交互式图表

作为一个熟悉数据更新、过渡和动画的老手，接下来应该考虑实现交互式的图表了。

10.1 绑定事件监听器

什么？我知道，我知道，绑定数据，已经够人受的了。现在我又要讲什么绑定事件监听器？（这正是 JavaScript 这门奇怪的语言如此受人追捧的原因。）

正如第 9 章所提到的，JavaScript 有一个事件模型，在这个模型中，“事件”由发生的事情来触发，比如用户通过键盘、鼠标或触摸屏输入信息。大多数情况下，没人监听事件，它们会自生自灭。

为了让图表具有交互能力，我们必须针对一些事件来编写代码，以便监听某些 DOM 元素发生的这些事件。第 9 章曾写过这样的代码：

```
d3.select("p")
    .on("click", function() {
        //单击后执行一些操作
    });
```

这样就给 p 元素绑定了一个事件监听器。这个监听器用于监听单击（click）事件，也就是用户鼠标单击 p 元素时触发的事件。（D3 不使用自定义事件，但你自己可以定义。为了兼容既有标准，D3 支持所有 JavaScript 事件，比如 mouseover 和 click。而浏览器对事件的支持则参差不齐。建议大家参考 Peter-Paul：Koch 的事件兼容性表格，地址：http://www.quirksmode.org/dom/events/。）

这提醒我们，JavaScript 中的事件不会在真空里发生。事件总要在特定的元素上发生。因此上面的代码不会在任何 `click` 事件发生时都执行，而只会在单击 `p` 元素时才会执行。

虽然通过纯 JavaScript 代码也可以实现事件绑定，但 D3 的 `on()` 方法对于绑定 D3 元素集则特别方便。你也看到了，`on()` 接受两个参数：事件名和在元素上发生事件时要执行的匿名函数。

为图表赋予交互能力很简单，只要两步：

- 绑定事件监听器；
- 定义行为。

10.2　什么是行为

我们还是以前面实现的静态条形图为例，先看看示例页面 01_start.html。

前面示例代码只在一个元素 `p` 上绑定了事件监听器，这对 `on()` 来说是个不太常见的用法。更多的时候，我们会一次性为多个元素（比如图表中所有可见的元素）绑定事件监听器。这也不难，把只选择一个元素的 `select()` 换成选择多个元素的 `selectAll()`，再把选择的元素集交给 `on()` 就行了。

甚至，在一创建元素的时候，就可以为它绑定事件监听器。比如，下面是我们创建条形图中矩形的代码，我就在其末尾添加了一个 `on()`：

```
//创建条形
svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   ...  //设置属性（略掉）
   .on("click", function(d) {
        //任何条形被单击，都会执行这里的代码
   });
```

定义匿名函数时，可以引用 `d`、`i`，或二者都引用，一如平常。而位于这个函数中的代码，会在相应的单击事件发生时执行。

这也是验证数据值的一个简便方法，比如：

```
.on("click", function(d) {
             console.log(d);
});
```

试一试 02_click.html，打开 JavaScript 控制台，单击某个条形看看。单击条形时，应该能看到该条形的数据值被输出到控制台。确实方便！

悬停高亮

根据鼠标交互高亮元素是为图表赋予响应能力的一种常见方式，能帮助用户定位和关注相关的数据。

简单的悬停高亮只用 CSS 就能实现，根本不要 JavaScript！ CSS 的伪类选择符 `:hover` 可以跟其他选择符组合起来，表示同一个元素但现在鼠标指针正悬停其上的状态。比如，以下 CSS 规则会让所有 SVG 矩形在鼠标悬停时变成橙色，参见图 10-1：

```
rect:hover {
        fill: orange;
}
```

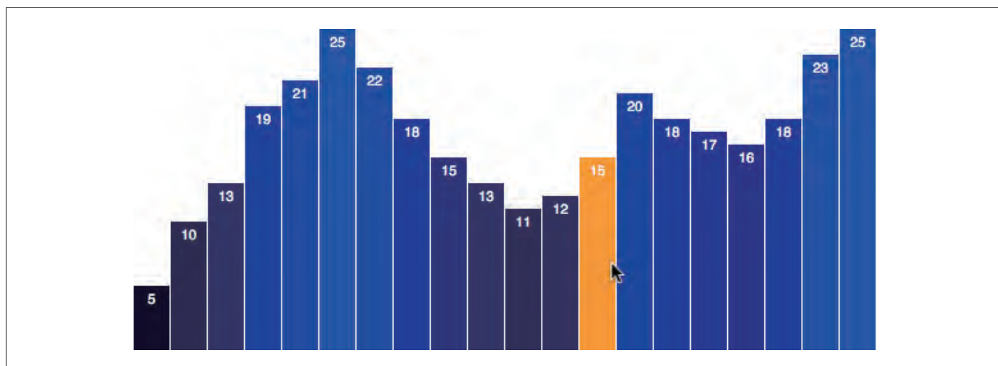


图 10-1：只用 CSS 实现的简单悬停效果

打开 03_hover.html，然后自己试一试。

CSS 悬停简便快捷，但有局限性。通过 `:hover` 只能做到改变样式。不过，最新的浏览器支持对 SVG 元素应用新的 CSS3 过渡。试试把下面的代码添加到上面 `rect:hover` 规则中：

```
rect {
        -moz-transition: all 0.3s;
        -o-transition: all 0.3s;
        -webkit-transition: all 0.3s;
        transition: all 0.3s;
}
```

这是在告诉浏览器（包括 Mozilla、Opera 和其他基于 Webkit 的浏览器），为 rect 元素的变化应用 0.2 秒的过渡效果。运行之后，你会看到蓝 / 橙切换不再是突变，而是会非常平滑地用 0.2 秒完成。确实不错！

不过，这些过渡也可以通过 JavaScript 和 D3 来控制，而且这样还能与图表其他部分协同。幸运的是，D3 把所有麻烦的事都替我们做了，因此写 JavaScript 代码也变得不那么痛苦了。下面就抛开 CSS 重新实现一下前面的悬停变色效果。

这次不能再使用 "click" 了，而要给 on() 传入 "mouseover"，这是与 CSS 的 :hover 等价的 JavaScript 事件：

```
.on("mouseover", function() {
        // 在鼠标悬停时执行这里的代码
});
```

好，我们想在用户鼠标悬停时把条形变成橙色。而我们现在的上下文是匿名函数，怎么选择发生事件的元素呢？

答案在 this。对不起，我说的确实是 this。只要选择 this，然后将其 fill 设置为 orange 即可：

```
.on("mouseover", function() {
        d3.select(this)
          .attr("fill", "orange");
});
```

“真正的”程序员之所以不喜欢 JavaScript，一个主要原因就是这个关键字 this。在其他语言中，this 的含义非常明确，不像 JavaScript 中那么多变。（jQuery 粉丝想必知道这方面的一些争论。）

对我们来说，只要知道以下几条就够了：

- 上下文很重要；
- 在匿名函数中，D3 把上下文自动设置为 this，因此 this 引用“我们要操作的当前元素”。

换句话说，在传给任何 D3 方法的匿名函数中，如果想操作当前元素，只要引用 this 就行了。

确实，你可以看到这一点，打开 04_mouseover.html（参见图 10-2）：

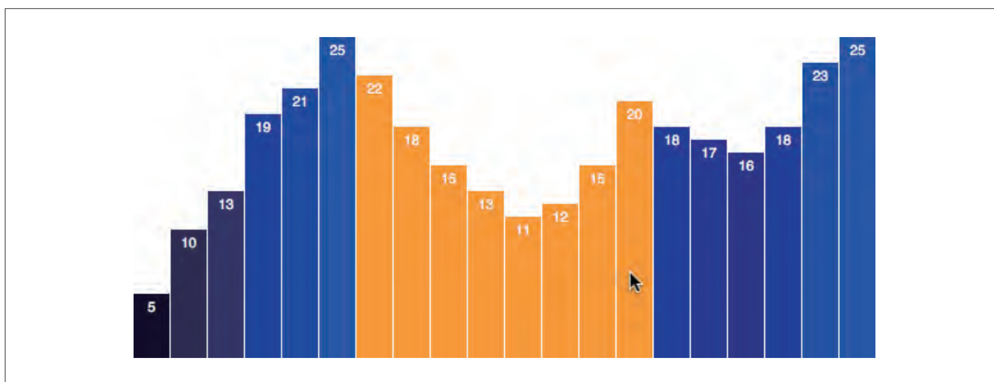


图 10-2：使用 D3 在鼠标悬停时设置填充颜色

把鼠标移到一个条形上，该条形上的事件监听器就会被触发，然后引用该条形的 this 被选中，代码将它的填充颜色设置为橙色。

图 10-2 看起来不错，但我们应该在鼠标离开条形时，再把填充变回原来的颜色。鼠标离开时会触发 mouseout 事件：

```
.on("mouseout", function(d) {
        d3.select(this)
      .attr("fill", "rgb(0, 0, " + (d * 10) + ")");
});
```

这下完美了！试一试 05_mouseout.html。参见图 10-3。

我太兴奋了，只用了 8 行 JavaScript 代码就实现了 CSS 用 3 行代码就能实现的效果！（不！）

实际上，我真正兴奋的原因是现在终于可以使用水润丝滑的过渡效果了（参见图 10-4）。还记得第 9 章的内容吗，实现过渡效果只要一个 transition() 和一个 duration() 即可：

```
.on("mouseout", function(d) {
    d3.select(this)
      .transition()
      .duration(250)
      .attr("fill", "rgb(0, 0, " + (d * 10) + ")");
});
```

打开 06_smoother.html 试一试。

图 10-3：左右移动鼠标，触发 mouseover 和 mouseout 事件

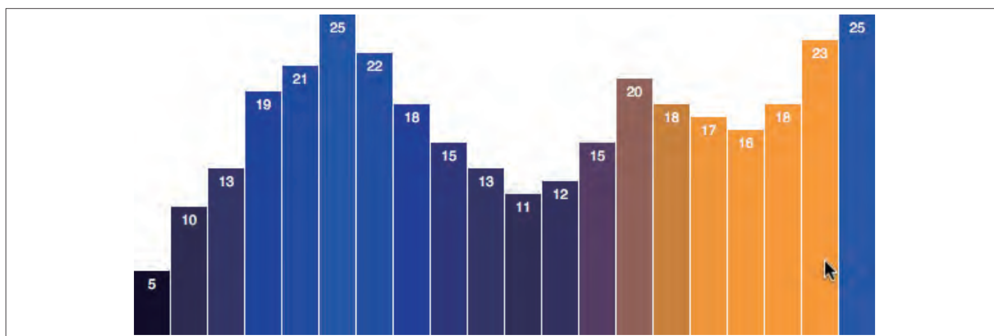


图 10-4：左右移动鼠标（水润丝滑版）

指针事件与重叠元素

鼠标事件只有在某个元素有像素被鼠标“接触到”的时候才会触发。如果两个元素叠加在一起，鼠标经过的元素只是“上面的”那个（即离你眼睛更近的那个）。这时候，只有上面的元素才会触发mouseover事件，下面的元素不会。

在06_smoother.html中可以验证这一点。把鼠标放到任意一个条形上，然后再把光标移到上方的值标签上。你会看到条形从橙色变成了蓝色。由于文本元素位于条形前面，因此鼠标移动到值标签上，就会触发下方条形的mouseout事件。但从视觉上看，鼠标并没有离开矩形区域，所以这好像有点违反直觉。但对于JavaScript而言，鼠标确实离开条形了。

记住：在SVG中，后加入DOM的元素在视觉层次上会被渲染在先加入元素的前面。（参见3.8.4节。）

有时候，我们希望在某些元素上（比如在值标签上）忽略鼠标事件。幸好，只要在CSS中给要忽略的元素添加一行代码即可：

```
pointer-events: none;
```

这行代码会通知浏览器：“嘿，对这个元素不要触发任何指针事件（比如click、mouseover或mouseout），就当这个元素不在那儿。”这样事件就可以穿透这个元素，在它下面的元素上触发。

要使用常规的CSS选择符来选择相应的元素。比如，下面这条规则将应用到所有SVG的text元素：

```
svg text {
        pointer-events: none;
}
```

如是不想在样式表中添加规则，也可以在D3中创建相应text元素时，直接设定这条CSS属性，比如：

```
svg.append("text")
        ...  // 其他代码
        .style("pointer-events", "none");
```

10.3 分组SVG元素

注意啦，g 元素自身不会触发任何鼠标事件。为什么呢？很简单，因为 g 元素没有像素！被它封装的元素，比如 rect、circle 和 text 元素才有像素。

但我们照样可以把事件监听器绑定给 g 元素。只要记住一点就行：被包含在 g 元素

中的元素给人感觉就是一个团队。比如，要是被包含的元素中有哪个被单击了或被鼠标悬停了，那么绑定到 g 元素的事件监听器也会被触发。

在一组视觉元素需要统一行为的情况下，这种机制相当有用。比如条形图吧，可以把 rect 和 text 元素分别放在各自的组里，原来元素的层次如下所示：

```
svg
        rect
        rect
        rect
        ...
        text
        text
        text
        ...
```

分组之后，就变成了这样：

```
svg
        g
                rect
                text
        g
                rect
                text
        ...
```

不用再担心什么指针事件，也不用管什么元素在上面，只要给整个组绑定事件监听器即可。这样，无论单击 rect 还是单击 text，都会触发相同的代码，因为它们在一个组里。

还有更厉害的呢，你可以在每个组上都放一个不可见的 rect，将其 fill 属性设置为 none，将其 pointer-events 属性设置为 all。即便这个 rect 不可见，它仍然可以触发鼠标事件，而且它的高度与图表区域的高度是一样的。结果就是鼠标经过条形的任何地方，包括短条形上方“空的”空白区域，都会触发颜色反转。

单击排序

交互式图表真正的强大之处，体现在能够展示数据的不同视图，吸引用户从不同角度来探索数据中蕴藏的奥秘。

对数据进行排序是非常重要的一种功能。而且正如你所料，D3 让排序元素变得异常简单。

仍以条形图为例，我们接下来给每个条形添加一个 click 事件监听器，在这个匿名函数中调用我们新定义的一个函数 sortBars()：

```
...
.on("click", function() {
        sortBars();
});
```

为简单起见，我们把它绑定到了每一个条形上面。当然，你完全可以把它绑定到一个按钮或其他元素上，无论该元素在不在 SVG 图形中。

在代码最后，定义新函数并将其保存到变量 sortBars 中：

```
var sortBars = function() {

        svg.selectAll("rect")
           .sort(function(a, b) {
                 return d3.ascending(a, b);
           })
           .transition()
           .duration(1000)
           .attr("x", function(d, i) {
                 return xScale(i);
           });

};
```

相关代码可以查看 07_sort.html，结果如图 10-5 所示。随便单击一个条形看看它们会不会重排顺序。

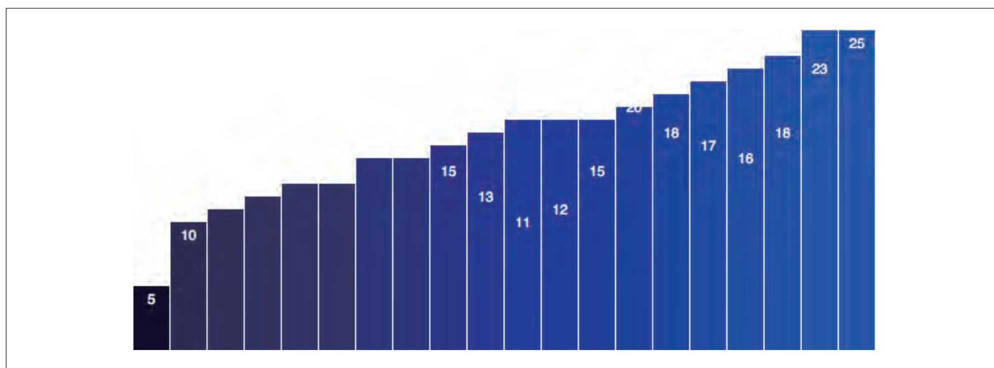


图 10-5：单击排序后的结果

单击调用 sortBars() 时，首先会选择所有 rect 元素。然后使用 D3 提供的 sort() 方法对它们进行排序，排序依据是绑定到它们的数据值。sort() 需要知道哪个元素排在前面，哪个元素排在后面，所以我们就传给了它一个比较函数。

与迄今为止我们看到的匿名函数不同，比较函数不接收 d（当前数据值）或 i（当前索引）作为参数，而是接受另外两个值：a 和 b。这两个值代表来自两个不同元

素的数据值。（给这两个参数换其他名字也没问题，这里用 a 和 b 也是一种惯例。）这个比较函数会针对数组中的每一对元素都被调用一次，然后它比较 a 和 b，直到所有数组元素都按我们指定的规则排序完毕。

在比较函数中，我们指定了 a 和 b 进行比较的规则。好在，D3 已经写好了很多比较函数，我们就省事了，不用每次都写很多 JavaScript 了。这里使用的是 d3.ascending()，而且把 a 和 b 直接传给了它。显然，哪个元素的值大，哪一个就胜出（返回）。sort() 就以这种方式循环所有数据值，直至所有元素都排序完毕。（注意，因为我们的数据都是数值，所以用 d3.ascending() 没问题。如果数据包含的是字符串，用它可能就乱套了。）

最后，排序工作完成，启动一个过渡，将持续时间设为 1 秒，然后计算每个 rect 的新 x 值。（这里调用 attr() 的代码是从一开始创建 rect 的代码那复制来的。）

目前来看，还有两个小问题。

首先，我们还没有考虑值标签呢，所以它们并没有跟条形一块滑到位。（这个要留给大家当练习了。）

其次，你可能已经注意到了，在过渡期间，如果把鼠标移动到某些条形上面，这些条形就不会排到正确的位置上（参见图 10-6）。

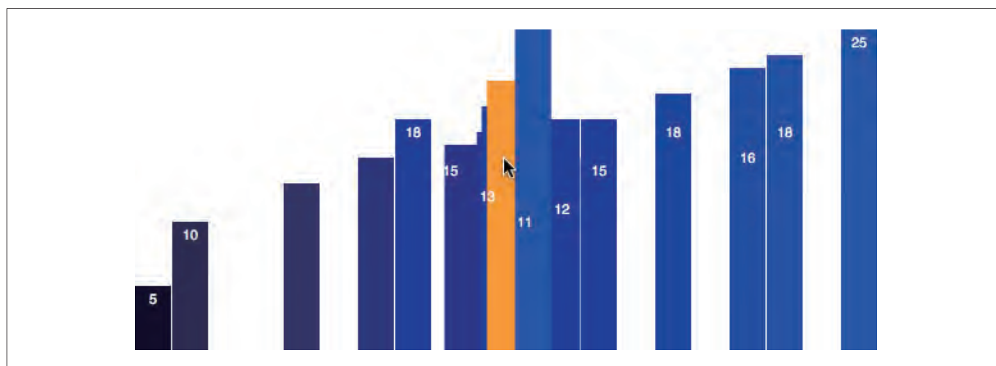


图 10-6：被打断的过渡

没错，看起来是不好看。

回想一下，第 9 章介绍过新过渡效果会打断和覆盖旧过渡效果。单击条形会触发一次过渡，而随后的鼠标悬停又会触发另一次过渡。为了运行鼠标悬停的高亮过渡，原来正在运行的过渡就会中断。结果就是那些被鼠标悬停过的条形永远不会到达它们的目标位置。

不用担心，这个例子只是为了说明悬停效果最好还是交给 CSS 打理，而那些视觉变化更密集的操作可以交给 D3 和 JavaScript。

在 08_sort_hover.html 里，我把高亮效果又还给了 CSS 负责，同时删除了 mouseover 和 mouseout 事件监听器，因此过渡冲突就不会再发生了。（唯一的遗憾是橙蓝渐变效果没有了。）

现在，排序还只是一个方向上的。我们可以实现再次单击触发重新排序，以降序排列条形。

要记住当前的条形图状态，可以再声明一个布尔变量：

```
var sortOrder = false;
```

然后，在 sortBars() 函数中，应该反转 sortOrder 的值，以便让它如果为 true 就会变成 false，反之亦然：

```
var sortBars = function() {

        // 反转 sortOrder 的值
        sortOrder = !sortOrder;

        ...
```

而在比较函数内部，可以再添加一点逻辑，判断如果 sortOrder 为 true，就按升序排序，否则按降序排序：

```
svg.selectAll("rect")
   .sort(function(a, b) {
                if (sortOrder) {
                        return d3.ascending(a, b);
                } else {
                        return d3.descending(a, b);
                }
        })
        ...
```

在浏览器中打开 09_resort.html 试一试。这回每次单击都会反转排序方式，如图 10-7 所示。

再增加一点效果会更酷。什么效果？每个元素逐个延迟过渡。（还记得什么是“目标一致性”吗？）

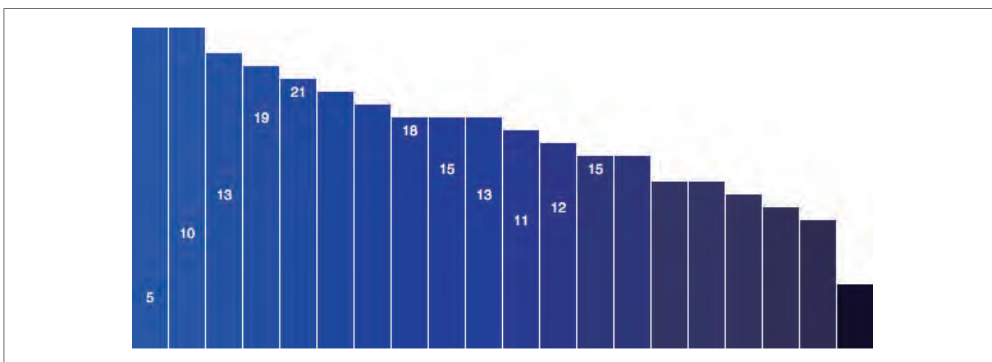


图 10-7：第二次排序，降序

你知道的，要实现这个效果，只需在 `transition()` 后面添加一个 `delay()` 语句即可：

```
...
.transition()
.delay(function(d, i) {
        return i * 50;
})
.duration(1000)
...
```

好吧，再看一看 10_delay.html。每次单击后排序，很容易就可以看到每个条形的运动轨迹。

10.4 提示条

在交互式图表中，提示条就是一个小的覆盖层，用于展示数据值。很多情况下，没有必要在默认的视图中为每个数值都加上标签，但又要确保这个层次的细节让用户有办法看到。此时就需要提示条。

本节，我们介绍三种在 D3 中创建提示的方法，有最简单的也有最复杂的。

10.4.1 浏览器默认提示条

这应该是你首先需要知道的。浏览器默认提示条简单，能起作用，但不好看。有些浏览器中的提示条，干脆就是难看的黄色框，只要鼠标静止时间足够长，它们就会现身。这种提示条很容易实现，而出现的位置则由浏览器决定，当然，它们的外观我们同样一点也控制不了。

图 10-8 展示了删除值标签、使用浏览器默认提示条的条形图。只要把鼠标放在任何条形上，过几秒钟就能看到提示条出来。

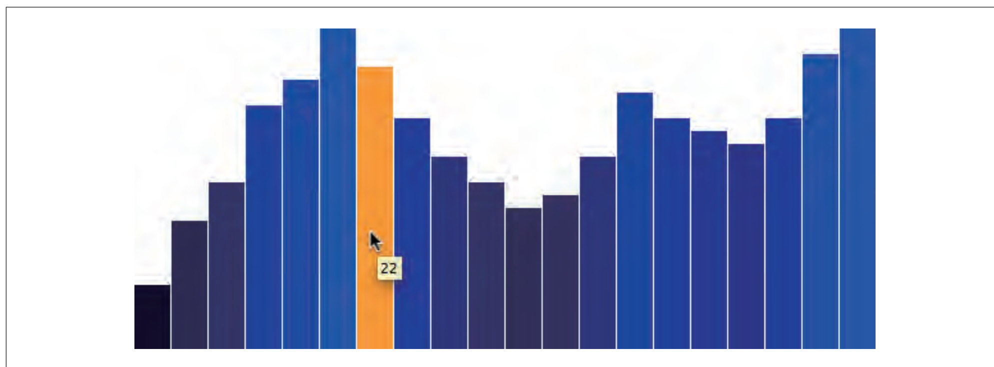


图 10-8：在 Safari 中看到的简单的浏览器默认提示条

代码详见 11_browser_tooltip.html。实现这样的提示条，只要在给那些需要提示条的元素嵌入一个 `title` 元素即可。例如，在创建所有 `rect` 元素后

```
svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   ...
```

可以直接把 `title` 元素追加进去：

```
   .append("title")
   .text(function(d) {
         return d;
   });
```

`append()` 会创建新的 `title` 元素，而接下来的 `text()` 会将其文本内容设置为 `d`，即绑定元素的值。

为了不让提示条显得太小，可以在内容前面添加一些文本（如图 10-9 所示）：

```
   .append("title")
   .text(function(d) {
         return "This value is " + d;
   });
```

完整的代码，请参见 12_browser_tooltip_text.html。

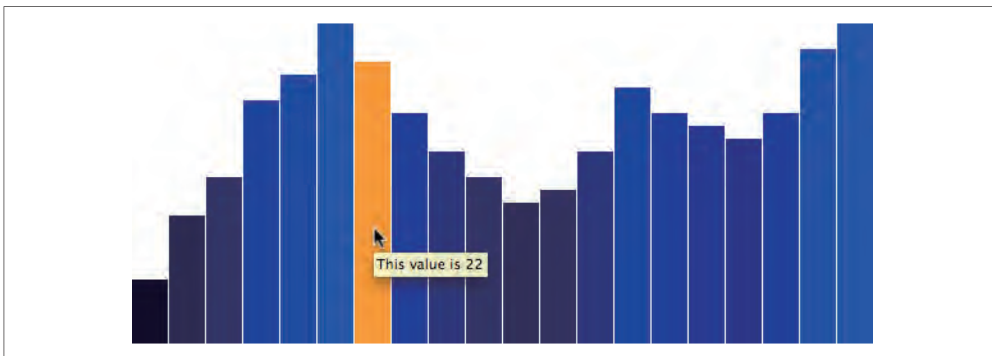


图 10-9：内容带前缀的浏览器默认提示条

10.4.2 SVG元素提示条

要想更多地控制提示条，就要将它们编码为 SVG 元素。

还是那句话，使用 SVG 元素作提示条的方案很多。我的方案是添加事件监听器，在每次发生 mouseover 事件时，动态创建值标签，而在 mouseout 事件发生时，将值标签删除。（或者，可以预先生成所有标签，而后根据鼠标悬停状态切换它们的显示和隐藏属性。再或者，只创建一个标签，然后根据需要显示 / 隐藏并改变其位置。）

回到我们的条形图，现在还得再把 mouseover 事件监听器添加回来。在这个匿名函数中，首先取得当前元素的 x 和 y 值（用 this，记得吧？），我们要用这两个值确定新提示条的位置，以便它恰好出现在触发事件的条形“上面”。

取得这两个值之后，分别把它们传给 parseFloat() 这个 JavaScript 函数：“嘿，就算这是个字符串，拜托也把它给我转换成浮点值。”

最后，把 x 和 y 都加大一些，以便让提示条位于条形顶部的中间：

```
.on("mouseover", function(d) {

//取得条形的x/y值，增大后作为提示条的坐标
var xPosition = parseFloat(d3.select(this).attr("x")) + xScale.
rangeBand() / 2;
var yPosition = parseFloat(d3.select(this).attr("y")) + 14;
```

这是最麻烦的地方。我们现在所做的一切，都是为了将提示条创建为一个简单的文本元素。当然啦，你也可以在这里添加一个 rect 作为背景，或者添加其他视觉效果：

```
//创建提示条
svg.append("text")
  .attr("id", "tooltip")
```

```
    .attr("x", xPosition)
    .attr("y", yPosition)
    .attr("text-anchor", "middle")
    .attr("font-family", "sans-serif")
    .attr("font-size", "11px")
    .attr("font-weight", "bold")
    .attr("fill", "black")
    .text(d);

})
```

没错，这是在以前值标签代码的基础上简单修改而来的。我们把 x 和 y 属性的值设定为刚刚计算得到的新位置值，把标签的实际文本设定为传递到事件监听器中的数据值 d。

同时要注意，我们给这个 text 元素指定了叫 tooltip 的 ID。这样必要时（mouseout），就容易选择（并删除！）该元素：

```
.on("mouseout", function() {

    //删除提示条
    d3.select("#tooltip").remove();

})
```

测试一下代码吧，在 13_svg_tooltip.html 里面。

如图 10-10 所示，使用 SVG 元素作为提示条，可以获得更新控制。要控制就得多花点时间。当然，我相信你可以做得比这个简单的例子更好。

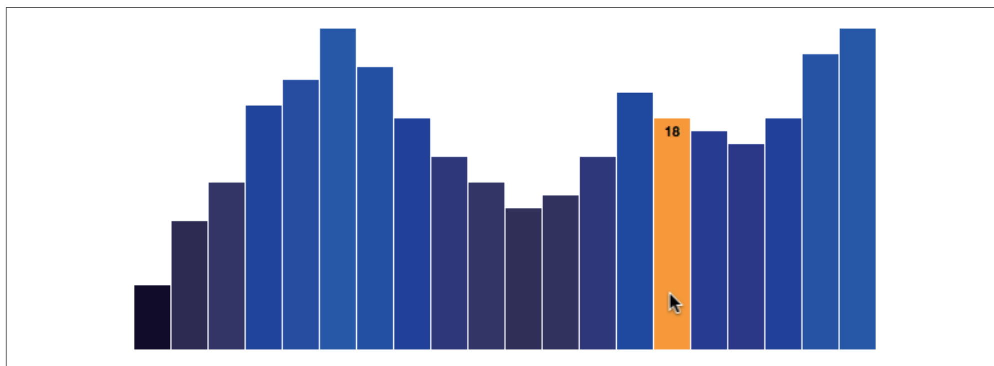


图 10-10：SVG 元素提示条

10.4.3　HTML的div提示条

与使用 SVG 元素类似，也可以使用 HTML 的 div 元素作为提示条，这适用于如下

情形：

- 实现的效果通过 SVG 不可能做到，或者支持不够好（例如 CSS 阴影）；
- 提示条要超出 SVG 图形的边界。

先看一看图 10-11 和图 10-12 给出的例子吧。

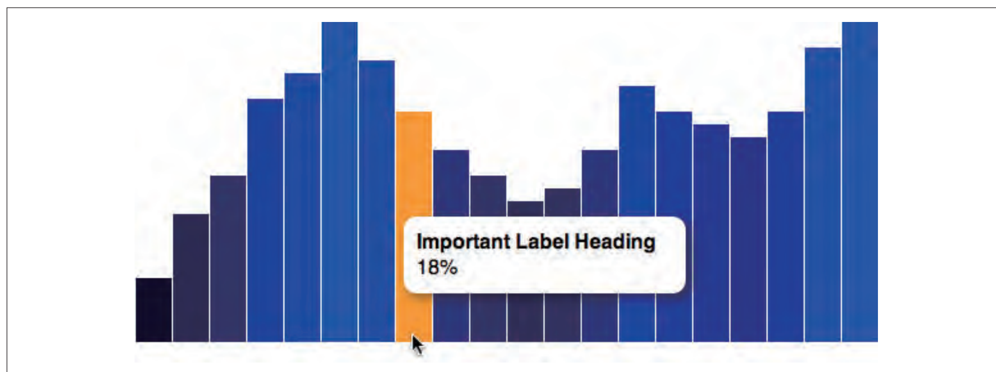


图 10-11：用 HTML 的 `div` 元素创建的提示条

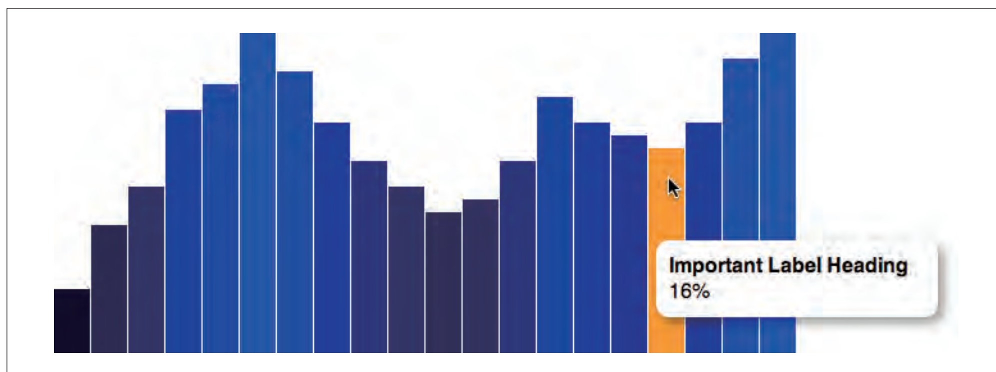


图 10-12：用 HTML 的 `div` 元素创建的提示条，摆脱了 SVG 图形的限制

还是那句话，实现方案不止一个。而我呢，喜欢在 HTML 里创建一个隐藏的 `div`，然后给它填充上值，在触发事件时再显示出来。最终代码可以参考 14_div_tooltip.html。

这个 `div` 可以使用 D3 来动态生成，而我喜欢手工输入：

```
<div id="tooltip" class="hidden">
        <p><strong>Important Label Heading</strong></p>
        <p><span id="value">100</span>%</p>
</div>
```

现在用 CSS 给提示条定义一些样式：

```
#tooltip {
        position: absolute;
        width: 200px;
        height: auto;
        padding: 10px;
        background-color: white;
        -webkit-border-radius: 10px;
        -moz-border-radius: 10px;
        border-radius: 10px;
        -webkit-box-shadow: 4px 4px 10px rgba(0, 0, 0, 0.4);
        -moz-box-shadow: 4px 4px 10px rgba(0, 0, 0, 0.4);
        box-shadow: 4px 4px 10px rgba(0, 0, 0, 0.4);
        pointer-events: none;
}

#tooltip.hidden {
        display: none;
}

#tooltip p {
        margin: 0;
        font-family: sans-serif;
        font-size: 16px;
        line-height: 20px;
}
```

特别注意 position 属性的值是 absolute，这样就可以精确控制它在页面上的位置了。而且，这里还添加了一些时髦的圆角和阴影。此外，pointer-events: none 可以保证鼠标经过提示条不会触发条形的 mouseout 事件，从而让提示条得以继续隐身。（试试没有这一行会怎么样，你就知道我说的是什么意思了。）最后，给提示条添加一个 hidden 类，它就隐身了。

另外，对 mouseover 事件监听器也要做一些改动，以便让这个 div 大致在触发它的条形上垂直居中。而且，根据 CSS 的布局需要，这里的代码也设定了提示条的 left 和 top 位置，将 ID 为 value 的 span 元素的文本内容设定为 d，然后——因为一切都就绪了——删除 hidden 类，让提示条现身：

```
.on("mouseover", function(d) {

// 取得条形的 x/y 值，增大后作为提示条的坐标
var xPosition = parseFloat(d3.select(this).attr("x")) + xScale.rangeBand() / 2;
var yPosition = parseFloat(d3.select(this).attr("y")) / 2 + h / 2;

// 更新提示条的位置和值
d3.select("#tooltip")
  .style("left", xPosition + "px")
  .style("top", yPosition + "px")
  .select("#value")
```

```
        .text(d);

    //显示提示条
    d3.select("#tooltip").classed("hidden", false);

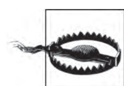
    })
```

在 mouseout 事件发生时隐藏提示条就简单多了，只要添加 hidden 类即可：

```
    .on("mouseout", function() {

        //隐藏提示条
        d3.select("#tooltip").classed("hidden", true);

    })
```



这个例子中的布局没有大问题，但现实中 D3 图表只是众多页面元素中的一部分，而完美的 HTML/CSS 布局仍然是很有挑战性的。这也是在 SVG 图表中采用 HTML 元素最大的问题。把提示条 div 和 SVG 图表都放到一个元素（如一个容器 div）中会好一些，因为这样只要关心相对位置就行了。d3.mouse 可以让你取得鼠标相对于页面上其他任何元素的坐标，对需要相对鼠标定位非 SVG 元素很有用。

10.5　适应触摸设备

iOS 和 Android 设备上的浏览器能够自动将触摸事件转换为鼠标事件，以方便 JavaScript 编程。换句话说，触摸某个元素会被浏览器解释为一次 click 事件。这意味着我们针对鼠标交互界面编写的代码，很大程度能在基于触摸的界面上使用。

主要的问题是多点触摸，D3 不能自动处理这些事件。虽然目前还没有处理多点触摸事件的简单方案，但 D3 能为我们跟踪触摸事件（至于怎么利用这些事件，就是你的事了）。要了解更多信息，请大家参考 d3.touches 的 API 文档吧，地址：https://github.com/mbostock/d3/wiki/Selections#wiki-d3_touches。

10.6　更进一步

恭喜！你现在已经掌握了 D3 所有的基本技能。绑定数据以及基于数据生成元素并设定样式、创建比例尺和绘制数轴、根据新数据修改已有元素、实现动画过渡，还有交互功能，这些已经让你成为 D3 的高手了。你还奢望什么呢？

布局和地图呢？好吧，接下来的两章我们就来介绍这两个更加高级的主题。等等，我得提前声明一下，这两章可没有前几章讲得那么详细。反正你已经掌握了基础知识了，那我何必多费口舌呢。开始吧！

第11章

布局

虽然叫“布局”，但大家可不能顾名思义。事实上，D3 不会对屏幕上的元素布什么局。D3 的布局方法没有直接的视觉输出，而是致力于把你提供的数据重新映射或变换成新格式，以便于在某些更特定的图表中的使用。数据有了新格式，怎么用或者生成什么图表，还是取决于你。

D3 全部的布局方法如下。

- Bundle：把霍尔顿（Holten）的分层捆绑算法应用到连线（edge）。
- Chord：根据矩阵关系生成弦形图（chord diagram）。
- Cluster：聚集实体生成系统树图（dendrogram）。
- Force：根据物理模拟定位链接的结点。
- Hierarchy：派生自定义的系统（分层的）布局实现。
- Histogram：基于量化的分组计算数据分布。
- Pack：基于递归圆形填充（circle packing）产生分层布局。
- Partition：递归细分结点树，呈射线或冰挂状。
- Pie：计算饼图或圆环图中弧形的起止角度。
- Stack：计算一系列堆叠的条形或面积图的基线。
- Tree：整齐地定位树结点。
- Treemap：基于递归空间细分来显示结点树。

本章只介绍三个最常用的布局方法：饼图（Pie）、堆叠（Stack）和力导向（Force）。这几种布局分别有不同的功能，不同的特性。

要想了解其他的 D3 布局方法，可以参考 D3 网站上的大量实例：https://github.com/mbostock/d3/wiki/Gallery，以及关于布局的 API 文档：https://github.com/mbostock/d3/wiki/Layouts。

11.1 饼图布局

`d3.layout.pie()` 可能没有它听起来那么可口，但还是值得关注。很明显，这个方法主要用于创建饼图，如图 11-1 所示。

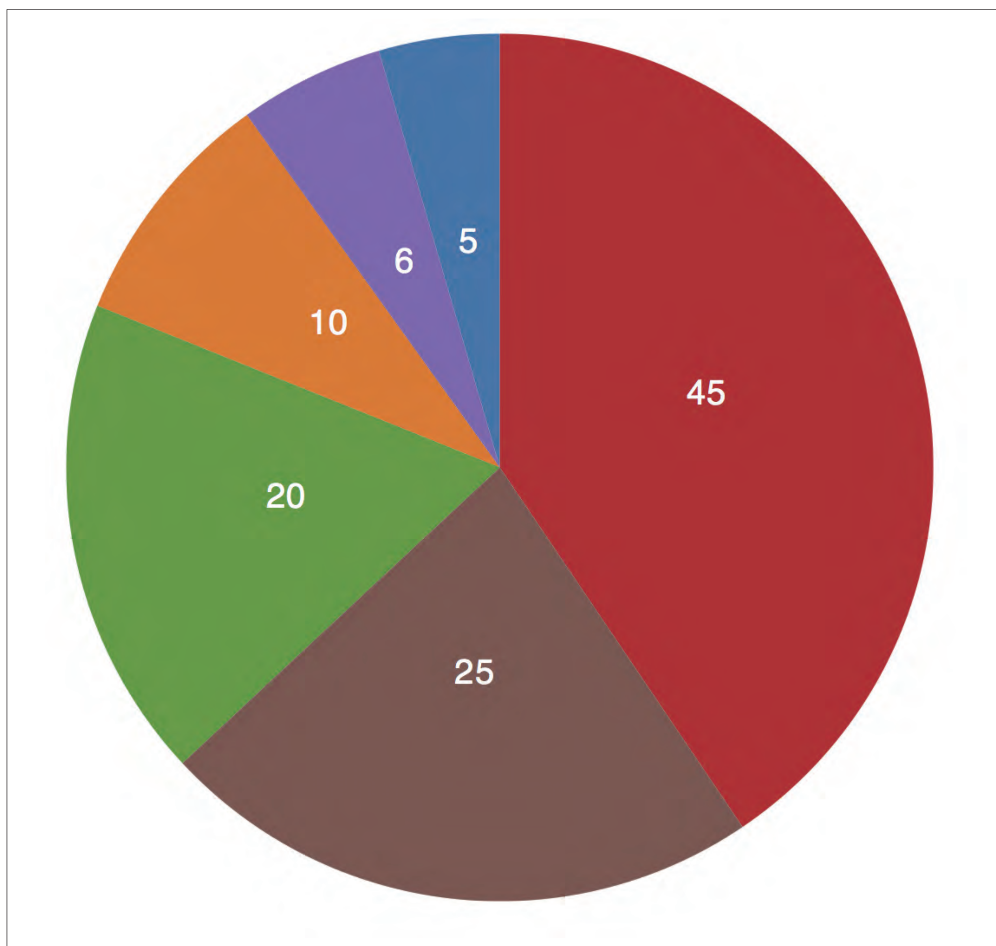


图 11-1：简单的饼图

大家可以打开 01_pie.html 随便看一看。

要想画出这些漂亮的扇形，必须知道几个数量值，包括每个扇形的内外半径和起止

角度。饼图布局的用途就是根据你的数据来计算出这些角度，让你根本用不着去回忆什么叫弧度。

还记得弧度？有读者记得，有读者可能早忘了。好吧，还是简单帮你回忆一下。一个圆有 360°，即 2π 弧度。因此，π 弧度等于 180°，也就是半圆。很多人都觉得度数好理解，但计算机则更喜欢弧度。

对于这个饼图，我们跟以前一样，还是使用了简单的数据集：

```
var dataset = [ 5, 10, 20, 45, 6, 25 ];
```

而要定义一个默认的饼图布局，那是相当简单：

```
var pie = d3.layout.pie();
```

然后，剩下的事儿就是把数据全都交给新创建的 `pie()` 函数，比如 `pie(dataset)`。下面比较一下转换前后的数据集吧，如图 11-2 所示。

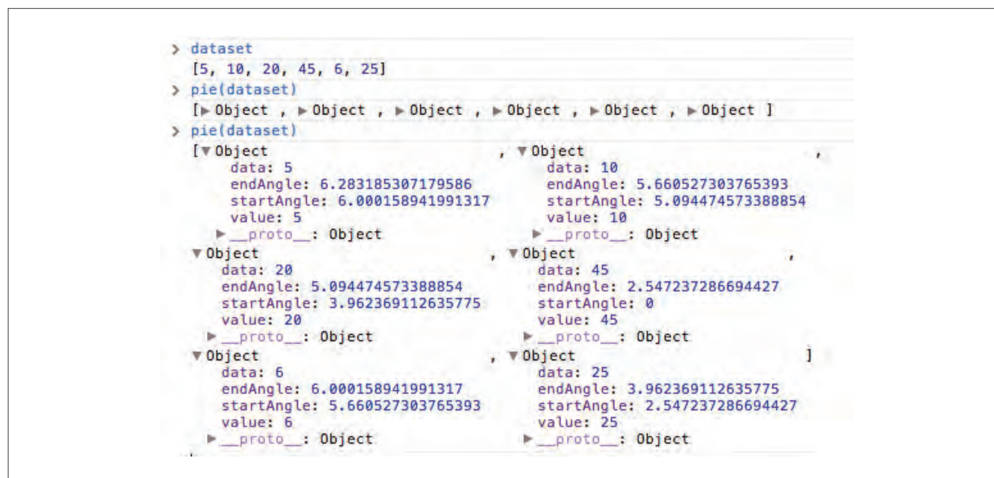


图 11-2：你的数据已经被转换成适合生成饼图的格式了

饼图布局方法把简单的数值数组转换成了对象的数组，每个值一个对象。每个对象又包含几个值，最重要的是 `startAngle` 和 `endAngle`。哇噢，原来就这么简单！

现在，要绘制扇形，还得使用 `d3.svg.arc()`，这是 D3 中用 SVG 的 `path` 元素绘制弧形的内置方法。前面我们并没介绍过 `path`，现在可以告诉你它是 SVG 用来绘制不规则图形的一个元素。任何不是矩形、圆形或其他基本形状的图形，都可以用 `path` 来绘制。问题在于，定义路径（`path`）值的语法对人类并不怎么友好。以下就是绘制图 11-1 中那个最大的红色扇形的代码：

```
<path fill="#d62728" d="M9.184850993605149e-15,-150A150,150 0 0,1
        83.99621792063931,124.27644738657631L0,0Z"></path>
```

假如你能看懂这些，那就不用看这本书了。

否则，你只要记住：用 `d3.svg.arc()` 这种方法替我们生成路径就好了。你可别想着自己手工来写这些东西。

弧形是一个自定义函数，接受内圆半径和外圆半径作为参数：

```
var w = 300;
var h = 300;

var outerRadius = w / 2;
var innerRadius = 0;
var arc = d3.svg.arc()
                .innerRadius(innerRadius)
                .outerRadius(outerRadius);
```

这里是把整个图表设置成 300 × 300 的正方形，然后把 `outerRadius` 设定为边长的一半（即 150 像素），把 `innerRadius` 设定为 0。稍后我们还会再提到这个 `innerRadius`。

现在已经可以绘制某些扇形了！首先，创建 SVG 元素，跟往常一样：

```
// 创建 SVG 元素
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);
```

接下来可以为每个要绘制的扇形创建新的分组（`g`），把用于生成饼图的数据绑定到这些新元素，并把每个分组平移到图表中心，好让路径出现在合适的位置上：

```
// 准备分组
var arcs = svg.selectAll("g.arc")
        .data(pie(dataset))
        .enter()
        .append("g")
        .attr("class", "arc")
        .attr("transform", "translate(" + outerRadius + ", " + outerRadius
            + ")");
```

注意，我们把新创建的 `g` 元素的引用保存到了变量 `arcs` 中。

最后，在每个 `g` 元素中，我们都追加一个 `path` 元素。路径相关属性的值都保存在 `d` 中，所以这里调用 `arc` 生成器，基于绑定到这个分组的数据来生成路径信息：

```
// 绘制弧形路径
```

```
arcs.append("path")
    .attr("fill", function(d, i) {
        return color(i);
    })
    .attr("d", arc);
```

噢，你可能奇怪，那些颜色是从哪儿来的？在 01_pie.html 里，可以看到这行代码：

```
var color = d3.scale.category10();
```

D3 支持一些生成分类颜色的方法。这些方法生成的颜色也许你不喜欢，但对于原型阶段随便生成可视化效果则是非常方便的。`d3.scale.category10()` 会创建一个序数比例尺和包括 10 种颜色的输出范围。（要了解这些颜色比例尺的详细信息，请参考相关维基：https://github.com/mbostock/d3/wiki/Ordinal-Scales。其中还介绍了 D3 采用的 Cynthia Brewer 的感知颜色校准色盘。）

最后，可以为每个扇形生成文本标签：

```
arcs.append("text")
    .attr("transform", function(d) {
        return "translate(" + arc.centroid(d) + ")";
    })
    .attr("text-anchor", "middle")
    .text(function(d) {
        return d.value;
    });
```

注意在 `text()` 中，我们引用的是 `d.value` 而不只是 `d`。因为绑定的是饼图数据，所以不能再引用原始数组中的元素（`d`），而要引用对象数组中的值（`d.value`）。

唯一陌生的地方是 `arc.centroid(d)`。啥意思这是？所谓图心（centroid ）就是通过计算得到的任何图形的中心点，无论是常规图形（比如正方形）还是极为不规则的图形（比如马里兰州的边境线）。在这里，`arc.centroid()` 是个超级有用的函数，它负责计算并返回任何弧形的中心点。然后，再把文本标签元素平移到每个弧形的中心点。这就是文本标签能够定位到每个扇形中心点的秘密。

额外提醒：还记得 `arc()` 要求一个 `innerRadius` 值吧？我们可以修改这个值，让它大于 0。这样，饼图就会变成图 11-3 所示的圆环图。

```
var innerRadius = w / 3;
```

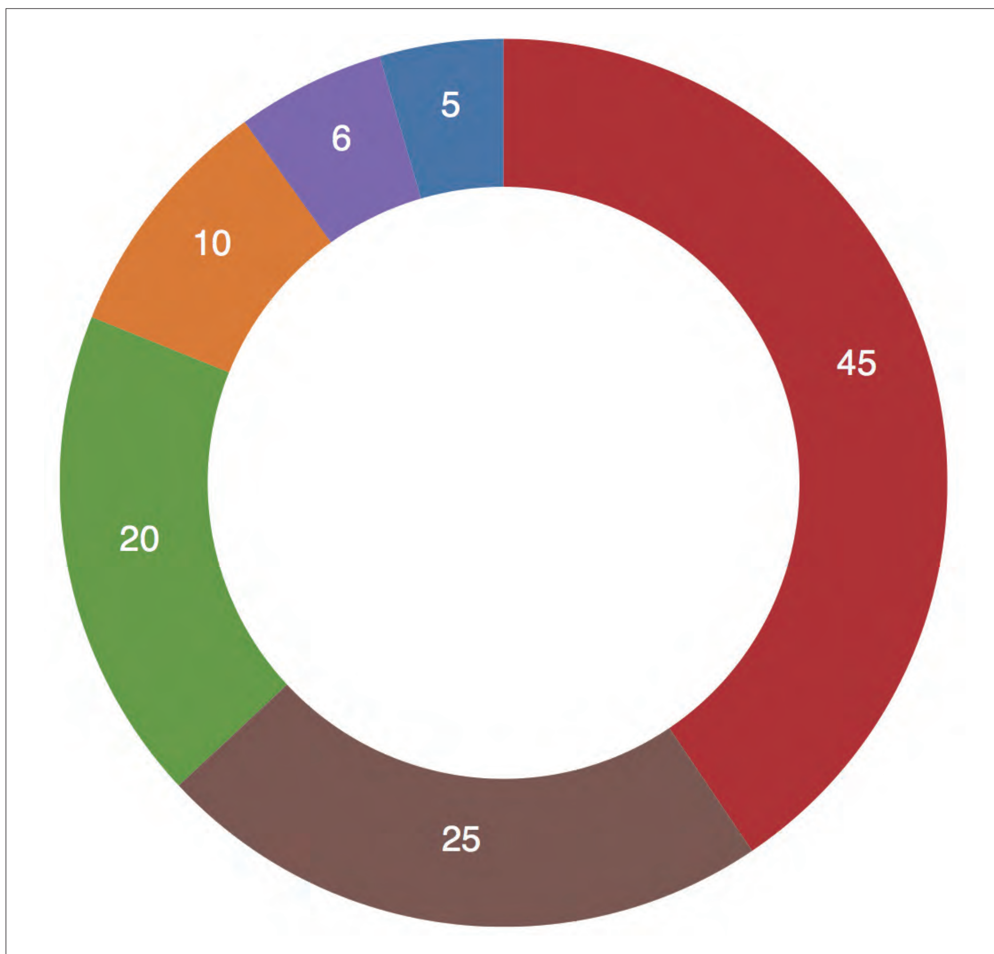


图 11-3：简单的圆环图

看看 02_ring.html 吧。

还有一点：饼图会自动按照从大到小的顺序排列。我们最初的数据集是 `[ 5, 10, 20, 45, 6, 25 ]`，因此你可能会认为代表 6 的扇形会位于代表 45 和 25 的扇形中间，其实不然。布局方法会按照降序对值进行排序，结果饼图的 12 点方向是代表 45 的扇形，其余扇形依次顺时针排列。

11.2 堆叠布局

`d3.layout.stack()` 能够把二维数据转换成“堆叠”数据，它会计算每个数据点的基线值，以便把数据层相互堆叠起来。这个布局方法可用于创建堆叠条形图、堆

叠面积图，甚至河流图（stream graph，就是没有严格零起点基线值的堆叠面积图）。

下面我们以图 11-4 所示的堆叠条形图为例。

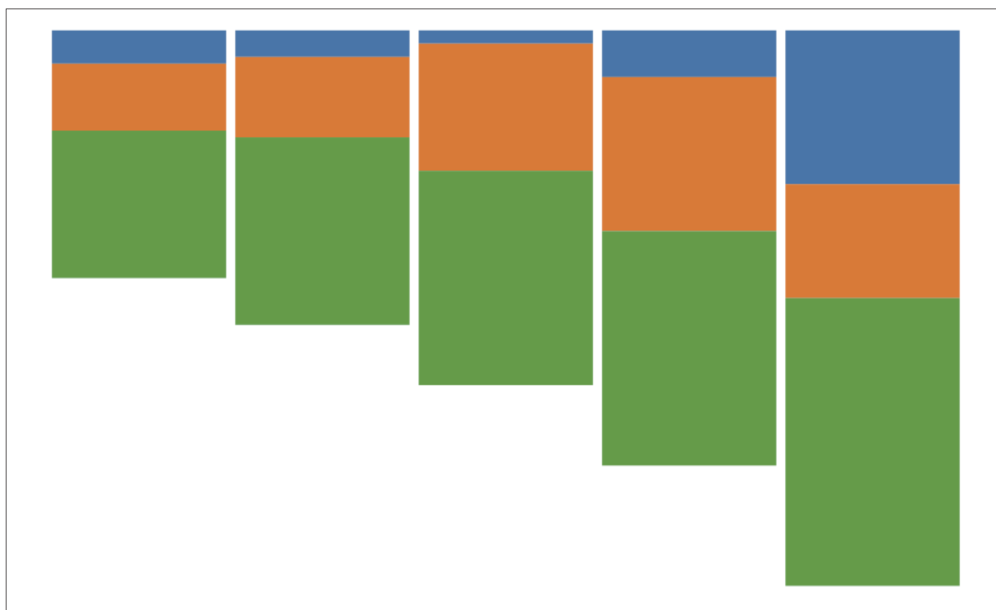


图 11-4：简单的堆叠条形图

假设有以下数据：

```
var dataset = [
        { apples: 5, oranges: 10, grapes: 22 },
        { apples: 4, oranges: 12, grapes: 28 },
        { apples: 2, oranges: 19, grapes: 32 },
        { apples: 7, oranges: 23, grapes: 35 },
        { apples: 23, oranges: 17, grapes: 43 }
];
```

第一步就要把这些数据重新组织成一个数组的数组，每个数组代表一个类别（如 apples、oranges 或 grapes）。在每个类别数组中，要用对象表示每个数据值，而表示每个数据值的对象本身包含 x 和 y 值。这里的 x 值其实就是 ID 值，而 y 才是实际的数据值：

```
var dataset = [
        [
                { x: 0, y: 5 },
                { x: 1, y: 4 },
                { x: 2, y: 2 },
                { x: 3, y: 7 },
```

```
                { x: 4, y: 23 }
        ],
        [
                { x: 0, y: 10 },
                { x: 1, y: 12 },
                { x: 2, y: 19 },
                { x: 3, y: 23 },
                { x: 4, y: 17 }
        ],
        [
                { x: 0, y: 22 },
                { x: 1, y: 28 },
                { x: 2, y: 32 },
                { x: 3, y: 35 },
                { x: 4, y: 43 }
        ]
];
```

原始数据集在控制台中如图 11-5 所示。

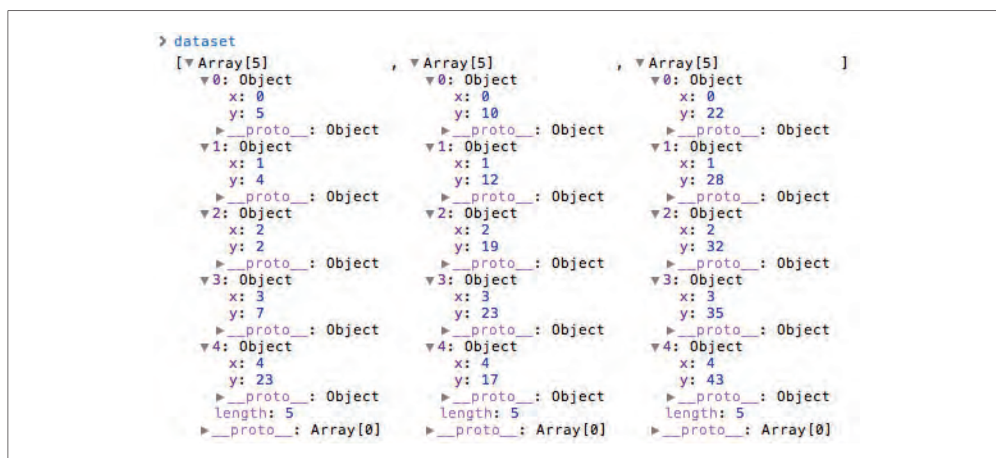


图 11-5：堆叠前的数据

然后，初始化一个堆叠布局函数，把原始数据集传进去：

```
var stack = d3.layout.stack();
stack(dataset);
```

现在，堆叠后的数据如图 11-6 所示。

能发现差别吗？在堆叠后的数据中，每个对象都被赋予了一个 y0 值。这就是基线值。仔细看一看会发现，这个 y0 基线值等于前面类别所有 y 值之和。比如，上面从左到右第一个对象的 y 值是 5，其 y0 值是 0。往右（oranges），第二列第一个对象的 y 值为 10，而 y0 值为 5（啊哈！就是第一列第一个对象的 y 值！）最右一

列（grapes），第一个对象的 y 值为 22，y0 值为 15（就是 5+10）。

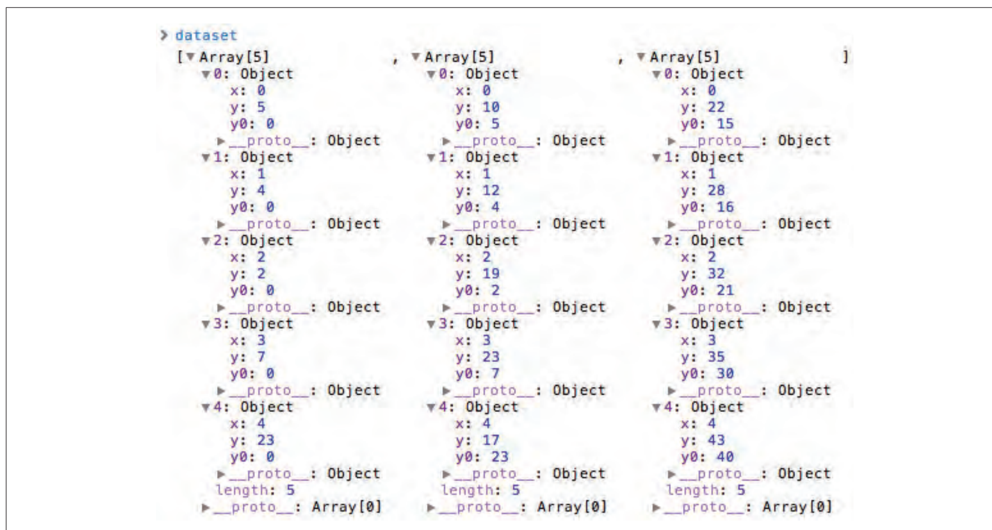


图 11-6：堆叠后的数据

为实现元素在视觉上的“堆叠”，我们需要引用每个数据对象的基线值，还有它的高度值。代码请参见 03_stacked_bar.html。（请大家找时间练习一下以底部的 *x* 轴为基础实现条形堆叠。）

```
var rects = groups.selectAll("rect")
        .data(function(d) { return d; })
        .enter()
        .append("rect")
        .attr("x", function(d, i) {
                return xScale(i);
        })
        .attr("y", function(d) {
                return yScale(d.y0);
        })
        .attr("height", function(d) {
                return yScale(d.y);
        })
        .attr("width", xScale.rangeBand());
```

注意在设置 `y` 和 `height` 时引用的分别是 `d.y0` 和 `d.y`。

11.3　力导向布局

之所以叫力导向（force-directed）布局，是因为这种布局模拟了物理学中的力在屏幕上排列元素的机制。尽管这种布局被用得有点滥，但说实话它确实太酷了。几乎

所有人都想学着做一个，因此本节我们就来讨论它。

力导向布局典型地要使用网状数据。在计算机科学领域，这种数据集叫做图（graph）。图由一组结点（node）和连线（edge）构成。结点代表数据集中的实体，连线代表结点之间的关系。有些结点可能有连线，而有些结点可能没有连线。结点一般以圆形表示，连线就是线。当然，具体用什么元素取决于你。D3 只负责后台实现。

从物理角度看，这种布局表现为粒子之间的互斥作用，但同时也由弹簧连接。互斥力会使粒子相互远离，避免在视觉上重叠，而弹簧可以防止它们离得太远，保证我们能在屏幕上看到它们。

图 11-7 是我们下面将要实现的例子。



图 11-7：简单的力导向布局

D3 的力导向布局需要我们分别提供结点和连线，而且都是对象数组的形式。下面就是一个包含 nodes 和 edges 元素的 dataset 对象，每个元素本身都是对象数组：

```
var dataset = {
        nodes: [
                { name: "Adam" },
                { name: "Bob" },
                { name: "Carrie" },
                { name: "Donovan" },
                { name: "Edward" },
                { name: "Felicity" },
                { name: "George" },
                { name: "Hannah" },
                { name: "Iris" },
                { name: "Jerry" }
        ],
        edges: [
                { source: 0, target: 1 },
                { source: 0, target: 2 },
                { source: 0, target: 3 },
```

```
                    { source: 0, target: 4 },
                    { source: 1, target: 5 },
                    { source: 2, target: 5 },
                    { source: 2, target: 5 },
                    { source: 3, target: 4 },
                    { source: 5, target: 8 },
                    { source: 5, target: 9 },
                    { source: 6, target: 7 },
                    { source: 7, target: 8 },
                    { source: 8, target: 9 }
          ]
};
```

一如往常，D3 不管你在这些对象中保存了什么数据。我们这里的结点（`nodes`）就是人名，而连线（`edges`）则包含两个值：来源（`source`）ID 和目标（`target`）ID。这些 ID 对应着上面的结点，比如 ID 为 3，表示 `Donovan`。如果是 3 与 4 连接，则表示 `Donovan` 与 `Edward` 连接。

对于生成力导向布局而言，以上数据可以说是再简单不过了。你还可以添加更多信息，而且 D3 实际上也会在我们提供的数据基础上额外添加很多数据，稍后我们就会看到。

下面我们看一看怎么初始化力导向布局：

```
var force = d3.layout.force()
                     .nodes(dataset.nodes)
                     .links(dataset.edges)
                     .size([w, h])
                     .start();
```

同样，这也是最低配置。在这里，我们指定要使用的结点和连线，指定了有效空间大小，还在配置好之后调用了 `start()`。

这样就能生成默认的力导向布局，但默认布局并不适合所有数据集。你猜对了，D3 提供了很多自定义选项。详细信息请查看相应的 API 文档：https://github.com/mbostock/d3/wiki/Force-Layout。在此，我们增大 `linkDistance`（连接结点的连线的长度），以及结点之间的负电荷（`charge`），以便它们相互把对方排斥得更远。（这样做不太讲人性，但我只是想让它们多分开点而已。）

```
var force = d3.layout.force()
                     .nodes(dataset.nodes)
                     .links(dataset.edges)
                     .size([w, h])
                     .linkDistance([50])        // <-- 新代码!
                     .charge([-100])            // <-- 新代码!
                     .start();
```

接下来，创建作为连线的 SVG 直线：

```
var edges = svg.selectAll("line")
        .data(dataset.edges)
        .enter()
        .append("line")
        .style("stroke", "#ccc")
        .style("stroke-width", 1);
```

注意，我们把所有线条都设置成了相同颜色和宽度，而实际上是可以根据数据来动态设置这些属性的（比如，对“强”连接可以使用较粗或较宽的连线。）

然后，为每个结点创建 SVG 圆形：

```
var nodes = svg.selectAll("circle")
        .data(dataset.nodes)
        .enter()
        .append("circle")
        .attr("r", 10)
        .style("fill", function(d, i) {
                return colors(i);
        })
        .call(force.drag);
```

我们把所有圆形的半径都设置为相等，但颜色各不相同，这样做只是为了好看而已。当然，这些值可以动态设置，从而让图形更有利用价值。

这里要看一看最后一行代码，这是在启用拖放交互方式。（如果把这一行注释掉，用户就不能拖动结点了。）

最后，还必须指定在这个力导向布局“打点”（tick）的时候会发生什么。在物理模拟中，“打点”用于代指经过一段时间，就像时钟上的秒针走过一格一样。如果动画每秒钟要播放 30 帧，那么打一次点就可以表示 1/30 秒。这样，模拟程序每打一点，就可以实时看到动画更新的计算过程。在某些应用中，打点可以比实际快一些。比如，如果建模对象是地球 50 年来的气候变化的影响，那你肯定不想等 50 年再看到结果，因此就要提前设置模拟系统的打点，让它比实际时间快。

对我们这个例子来说，只需要知道 D3 的力导向布局能像其他物理模拟系统一样超越时间向前“打点”。每打一次点，力导向布局就会根据初始化布局时指定的数据，调整每个结点和连线的位置值。要亲眼看到这个过程，就得更新与之关联的元素——直线和圆形：

```
force.on("tick", function() {

    edges.attr("x1", function(d) { return d.source.x; })
         .attr("y1", function(d) { return d.source.y; })
         .attr("x2", function(d) { return d.target.x; })
         .attr("y2", function(d) { return d.target.y; });
```

```
        nodes.attr("cx", function(d) { return d.x; })
             .attr("cy", function(d) { return d.y; });

    });
```

这是在告诉 D3："好，每打一次点，取得每条直线和每个圆形的新 *x*/*y* 值，在 DOM 中更新它们。"

稍等，这些 *x*/*y* 值是从哪来的？我们只指定了人名、来源和目标啊！

D3 会计算这些 *x*/*y* 值，并将它们追加到原始数据集中既有的对象上（参见图 11-8）。找到并打开 04_force.html，在控制台中输入 `dataset`。点开任何结点或连线，都能看到我们并没有提供的很多信息。这是 D3 为了继续执行物理模拟保存的信息。调用 `on("tick", ...)` 的时候，我们只要指定怎么取得更新后的坐标，然后将它们映射到 DOM 中的可见元素即可。

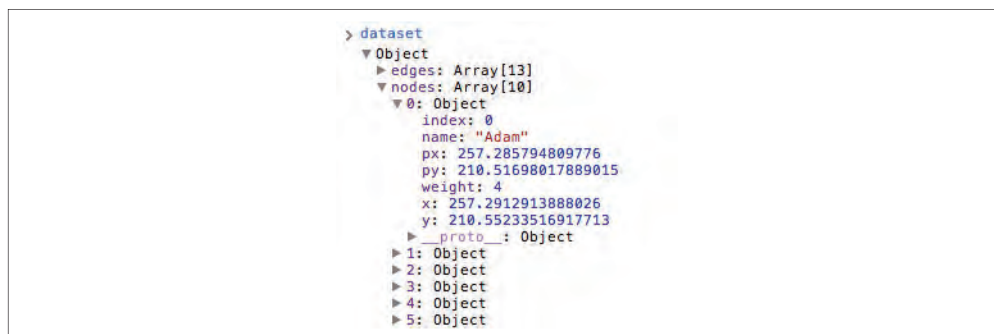


图 11-8：数据集中的第一个结点，包含 D3 补充的很多信息

那么最终结果，就是纠缠在一起圆形和线条，如图 11-9 所示。



图 11-9：包含 10 个结点和 12 条连线的简单力导向布局

要查看最终代码，可以看看 04_force.html。

注意，每次刷新页面，这些圆形和直线都会焕发生机，即使相对平衡的状态每次也不一样。之所以每次的静止状态都不一样，是因为存在一个随机元素，它决定了圆点以什么方式进入场景。这暗示了力导向布局不可预测的天性：这种布局每次都可能大不相同，而布局本身有赖于数据提供的结构信息。如果数据具有较强的结构性，视觉效果也会更好。

交互性对于改善数据视图的可用性非常有好处。比如，如图 11-10 所示，我不喜欢这里连线的摆布方式，可以把粉红色的圆形拖出来。然后再把红色的圆形向左向上拖一拖，如图 11-11 所示。



图 11-10：拖动结点改变整体布局



图 11-11：拖动其他结点，理顺布局

多说一句：交互性 + 物理模拟 = 难以抗拒的演示。我没法解释，但真是这样。不知何故，人类总是喜欢看到现实中的物体被呈现在屏幕上。

第 12 章

地图

条形图、散点图、圆环图，甚至还有力导向图……好，都很好。你想想，现在是不是该学习一下地图了?!

12.1 JSON与GeoJSON

我们已经认识 JOSN 了，现在再认识一下 GeoJSON。GeoJSON 是基于 JSON 的、为 Web 应用而编码地理数据的一个标准。实际上，GeoJSON 并不是另一种格式，而只是 JSON 非常特定的一种使用方法。

在学习创建地图之前，必须先取得要显示的形状的路径数据（轮廓）。我们就举一个最典型的例子：绘制美国的州界。本书示例代码中有一个文件叫 us-states.json。这个文件是直接从 D3 的一个示例中下载过来的，我们应该向 Mike Bostock 道声谢，感谢他准备了这么细致的州界文件。

打开 us-states.json，会看到如下内容（这里已经重排了格式并删除了很多内容）：

```
{
  "type": "FeatureCollection",
  "features": [
     {
       "type": "Feature",
       "id": "01",
       "properties": { "name": "Alabama" },
       "geometry": {
         "type": "Polygon",
```

```
            "coordinates": [[[-87.359296,35.00118],
              [-85.606675,34.984749],[-85.431413,34.124869],
              [-85.184951,32.859696],[-85.069935,32.580372],
              [-84.960397,32.421541],[-85.004212,32.322956],
              [-84.889196,32.262709],[-85.058981,32.13674] ...
             ]]
          }
        },
        {
            "type": "Feature",
            "id": "02",
            "properties": { "name": "Alaska" },
            "geometry": {
              "type": "MultiPolygon",
              "coordinates": [[[[-131.602021,55.117982],
                [-131.569159,55.28229],[-131.355558,55.183705],
                [-131.38842,55.01392],[-131.645836,55.035827],
                [-131.602021,55.117982]]],[[[-131.832052,55.42469],
                [-131.645836,55.304197],[-131.749898,55.128935],
                [-131.832052,55.189182], ...
               ]]]
             }
          }
    ...
```

典型的 GeoJSON 数据，乍一看，一定是一个巨大的对象。（还记得花括号吗？）这个对象有一个值为 `FeatureCollection` 的 `type` 属性，还有一个 `features` 属性。后者是一个数组，包含个别的 `Feature`（地理特征）对象。每个 `Feature` 对象代表美国的一个州，看它的 `properties` 属性，那里就包含着州名。

但 GeoJSON 文件中真正有价值的东西，还是 `geometry` 属性。这个属性下面包含着地理特征的类型（`type`），随后是构成地理边界的许多坐标。在 `coordinates` 对象中，是很多经度 / 纬度值对，每一对经纬度值都是一个小数组。这就是测绘人员耗尽心血得到的地理坐标数据。在这里感谢一代又一代的探险家和研究者，感谢他们为我们取得了这些虽然微小但又极其强大的数据资料。注意，经度是每个小数组的第一个元素。因此，尽管英语国家的人习惯说纬度 / 经度，但 GeoJSON 数据中则都是先经度后纬度。

另外，为了防止你对制图知识太生疏，我们再稍微恶补一点地理常识：

- 经代表纵向，因此经线是南北走向的，就像是从我们头顶上下来的一样；
- 纬代表横向，因此纬线是东西走向的，就像是地球的腰带一般。

经线和纬线一起交织成了巨大的网格，环绕着整个地球。而且，经度和纬度恰好能方便地转换为屏幕显示的 *x* 和 *y* 坐标。在条形图中，我们就是把数据值映射为显示值，即矩形的高度值的。在地图中，同样要把数据值映射为显示值，即经 / 纬度变

成 *x*/*y*。把经 / 纬度想象成 *x*/*y* 可以避免对经度在前纬度在后的不适。



Get Lat+Lon（http://teczno.com/squares）是用于复核坐标值的很好的资源，作者是 Michal Migurski。建议大家在绘制地图的时候，在另一个标签页中打开这个网站，以备不时之需。

12.2 路径

既然已经有了地理数据，那就动手吧。

首先，定义第一个路径生成器：

```
var path = d3.geo.path();
```

`d3.geo.path()` 完全是一个救苦救难的观世音函数，它在后台帮我们把乱七八糟的 GeoJSON 坐标转换成更乱的 SVG 路径代码。`d3.geo.path()` 万寿无疆！

现在倒可以把所有 GeoJSON 直接放到 HTML 文件里了——可是，那么多坐标和花括号，太乱了！常见的也是更清爽的办法是把地理数据保存一个单独的文件中，然后使用 `d3.json()` 来加载：

```
d3.json("us-states.json", function(json) {

        svg.selectAll("path")
           .data(json.features)
           .enter()
           .append("path")
           .attr("d", path);

});
```

`d3.json()` 接受两个参数。第一个是要加载的文件的路径，第二个是在加载并解析完 JSON 文件后执行的回调函数。（参见 5.2.2 节“处理数据加载错误”，了解关于回调函数的细节。）`d3.json()` 与 `d3.csv()` 类似，都是异步执行的函数。换句话说，浏览器在等待外部文件加载时，其他代码照样执行，不会受影响。因此，位于回调函数后面的代码有可能先于回调函数本身执行：

```
d3.json("someFile.json", function(json) {
        // 这里的代码依赖加载的 JSON
});

// 这里的代码不能依赖 JSON
console.log("I like cats.");
```

记住这条规则：在加载外部文件时，把依赖数据的代码放到回调函数中。（或者把代

码放到其他自定义函数中，然后在回调函数中调用这些函数。）

回到例子中来。最后，我们把 GeoJSON 的地理特征绑定到新创建的 path 元素，为每个特征值创建一个 path：

```
svg.selectAll("path")
    .data(json.features)
    .enter()
    .append("path")
    .attr("d", path);
```

注意最后一行，这里的 d（path 元素的数据属性）引用着路径生成器，这个生成器会像变魔术一样，取得绑定的地理数据并计算出所有 SVG 代码。结果如图 12-1 所示。

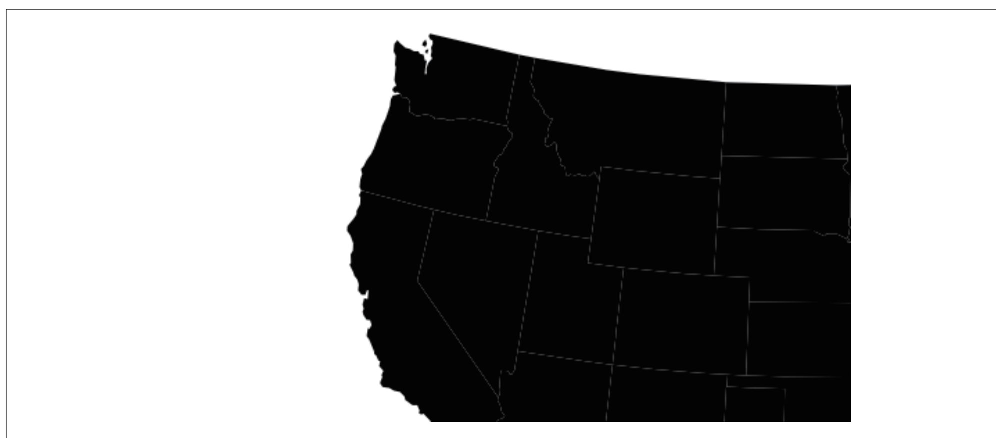


图 12-1：第一个基于 GeoJSON 数据生成的视图

地图！原来这么简单啊！打开 01_paths.html 看看吧。剩下的只是自定义的问题了。



关于路径和路径生成器的更多内容，请参考这里：https://github.com/mbostock/d3/wiki/Geo-Paths。

12.3 投影

聪明的读者一定注意到了，这张地图并没有覆盖美国全境。要纠正这个问题，需要修改我们使用的投影（projection）。

什么是投影？作为聪明的读者，你一定也注意到了，地球是圆的，不是平的。圆形的东西都是三维的，不适合在二维平面上表示。所谓投影，就是一种折中算法，一种把 3D 空间“投影”到 2D 平面的方法。

定义 D3 投影的方式我们很熟悉：

```
var projection = d3.geo.albersUsa()
                       .translate([w/2, h/2]);
```

D3 有几个内置的投影。Albers USA 是一种复合投影，可以把阿拉斯加和夏威夷整合到西南地区的下方。（马上就能看到效果了。）`albersUsa` 实际上是 `d3.path.geo()` 默认返回的投影。我们现在明确地指定它，可以设置几个自定义选项，比如平移的距离。这里是把投影平移到了 SVG 图形的中央（宽度的一半和高度的一半）。

现在唯一要修改的，就是明确告诉路径生成器，应该使用这个自定义的投影来生成所有路径：

```
var path = d3.geo.path()
                 .projection(projection);
```

这样就可以得到图 12-2 所示的结果。快点，打开 02_projection.html 看看代码吧。



图 12-2：同样的 GeoJSON 数据，但现在把投影居中了

还可以给投影添加一个 `scale()` 方法，把地图缩小一些，得到图 12-3 所示的结果。

```
var projection = d3.geo.albersUsa()
                       .translate([w/2, h/2])
                       .scale([500]);
```

默认的缩放值是 1000，比这个值小就会缩小地图，比这个值大就会扩大地图。

太酷了！代码请参考 03_scaled.html。

再添加一个 `style()` 语句，可以把路径的填充设置为没那严肃的颜色，比如图 12-4 中所示的蓝色。



图 12-3：缩小后的美国地图居中显示在了图形上

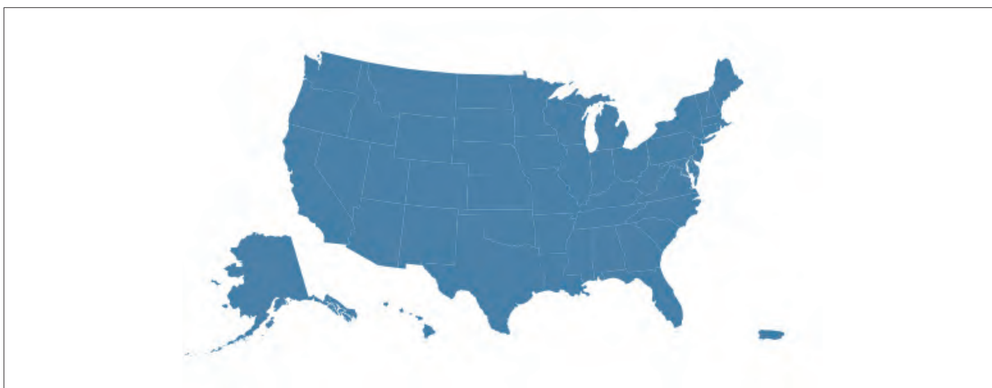


图 12-4：蓝色的地图比黑色看起来好多了

完成的代码在 04_fill.html 中。使用同样的方法，还可以设置描述颜色和宽度。

地图投影是极为强大的算法，不同的投影适合不同的用途和世界的不同地区（比如北极附近和赤道附近）。

主要应该感谢 Jason Davies，D3 的地理投影插件现在几乎可以支持任何你能想象得到的投影算法。关于 D3 的投影有一个完整的可视化参考，请访问其维基：https://github.com/mbostock/d3/wiki/Geo-Projections。另外，还有一个比较各种投影的演示，也很有参考价值：http://bl.ocks.org/3711652。

12.4　等值区域

等值……什么？这个词不太好念，它指的是不同区域填充了不同值（深或浅）或颜色，以反映关联数据值的地图。在美国，人们常说的“红州，蓝州”等值区域地图，

展示的是各个州对共和党和民主党的倾向，特别是在选举的时候很常见。但等值区域地图可以基于任何值而不光是党派倾向来生成。

这些地图也是使用 D3 生成最多的地图类型。尽管等值区域地图非常有实用价值，但也不要忘了它存在一些固有的感知局限性。因为这种图使用面积来编码值，人口密度低而面积大的州（比如内华达）从视觉上会显得比实际人数多。标准的等值区域图不能恰当地体现人均指标，内华达州太大了，而特拉华州又太小了。可是，这种图却能很好地标识地理区域，特别是在地图中，看起来真的非常非常酷。所以我们要好好探索一番。（可以参考 05_choropleth.html。）

首先，创建一个比例尺，将数据值作为输入，返回不同的颜色。这是等值区域地图的核心所在：

```
var color = d3.scale.quantize()
                    .range(["rgb(237,248,233)", "rgb(186,228,179)",
                    "rgb(116,196,118)", "rgb(49,163,84)","rgb(0,109,44)"]);
```

以量化的比例尺函数作为线性比例尺，但比例尺输出的则是离散的范围。这里输出的值可以是数值、颜色（这里就是），或者其他你需要的值。这个比例尺适合把值分类为不同的组（bucket）。我们这里只分了 5 个组，实际上你想分几个就分几个。

注意，这里只指定了输出范围，而没有指定输入值域。（我们得等到数据加载完毕后再做这件事。）这几个特别的颜色值来自 D3 托管在 GitHub 代码库中的 colorbrewer.js 文件（https://github.com/mbostock/d3/tree/master/lib/colorbrewer）。这个文件中包含了为人类感知优化的一批颜色值，由 Cynthia Brewer 根据她的研究选定。

接下来，要加载一些数据。我们有一个文件叫 us-ag-productivity-2004.csv，内容类似如下所示：

```
state,value
Alabama,1.1791
Arkansas,1.3705
Arizona,1.3847
California,1.7979
Colorado,1.0325
Connecticut,1.3209
Delaware,1.4345
...
```

这些数据来自美国农业部，报告内容是 2004 年每个州的农业生产力指标。这些值的单位是以 1996 年阿拉巴马州的生产力指标为基准（1.0），更大的值表示生产力更高，更小的值表示生产力更低。（更多美国政府公开的数据集可以这里找到：http://data.gov。）希望我们能利用这些数据生成一个漂亮的美国各州生产力地图。

要加载这些数据，使用 d3.csv()：

```
d3.csv("us-ag-productivity-2004.csv", function(data) { ...
```

然后，在回调函数中，要设置彩色的量化比例尺的输入值域（趁我还没忘！）：

```
color.domain([
        d3.min(data, function(d) { return d.value; }),
        d3.max(data, function(d) { return d.value; })
]);
```

这里用到了 d3.min() 和 d3.max() 来计算并返回最小和最大的数据值，因此这个比例尺的输出值域是动态计算的。

接下来，跟前面一样，加载 JSON 地理数据。但不同的是，在这里我想把农业生产力的数据合并到 GeoJSON 中。为什么？因为我们一次只能给元素绑定一组数据。GeoJSON 数据肯定必不可少，因为要据以生成路径，而我们还需要新的农业生产力数据。为此，就要把它们混合成一个巨大的数组，然后再把混合后的数据绑定到新创建的 path 元素。（混合数据有几种方法，这里使用我喜欢的方法。）

```
    d3.json("us-states.json", function(json) {

        //混合农业生产力数据和 GeoJSON
        //循环农业生产力数据集中每个值
        for (var i = 0; i < data.length; i++) {

            //取得州名
            var dataState = data[i].state;

            //取得数据值，并从字符串转换成浮点数
            var dataValue = parseFloat(data[i].value);

            //在 GeoJSON 中找到相应的州
            for (var j = 0; j < json.features.length; j++) {

            var jsonState = json.features[j].properties.name;

            if (dataState == jsonState) {

                //把数据值复制到 JSON 中
                json.features[j].properties.value = dataValue;

                //停止循环 JSON
                break;

        }
    }
}
```

一行一行地仔细看一遍代码。大致来说，就是对每个州，我们也找到 GeoJSON 中相应的值（如“Colorado”）。然后取得这个州的数据值，把它放到 json.

features[j].properties.value 中，保证它能被绑定到元素，并在将来需要时可以被取出来。

最后，像以前一样创建路径，只是通过 style() 要设置动态的值：

```
svg.selectAll("path")
    .data(json.features)
    .enter()
    .append("path")
    .attr("d", path)
    .style("fill", function(d) {
                //取得数据值
                var value = d.properties.value;

                if (value) {
                        //如果值存在……
                        return color(value);
                } else {
                        //如果值不存在……
                        return "#ccc";
                }
    });
```

这样，地图不再是千篇一律的“钢蓝色”了，每个州的路径都有了不同的填充值。这里有些小缺漏，因为我们并没有掌握所有州的数据。数据集里就缺少阿拉斯加、哥伦比亚特区、夏威夷和波多黎各（它虽不是一个州，但 GeoJSON 和投影中也包含它）等地的信息。

为了弥补这些缺漏，我们添加了一点小逻辑：一个 if() 语句，检查数据值是否有定义。如果数据值存在，就返回 color(value)，也就是说我们会把该数据值传递给量化比例尺，由比例尺返回颜色值。如果数据值不存在，则设置默认的浅灰色 #ccc。

非常漂亮！看看图 12-5 的结果吧。要想查看最终的代码，自己找一找 05_choropleth.html。

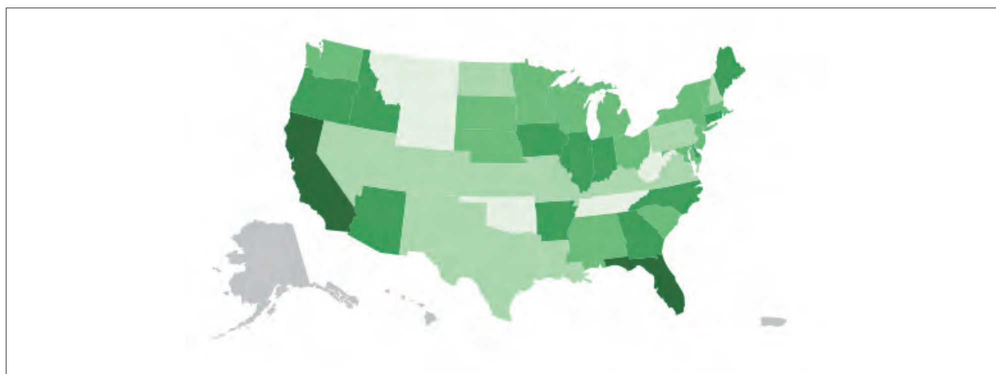


图 12-5：美国各州农业生产力等值区域地图

12.5 添加定位点

在地图上标出一些城市来是不是显得更真实呢？或许能够看到在那些生产力最高（或最低）的州有多少大城市会很有意思，也更有价值。同样，首先得把数据找来。

幸运的是，美国人口普查局为我提供了数据。（纳税人的税起作用了！）下面就是从美国人口普查局获得的“Annual Estimates of the Resident Population for Incorporated Places Over 50,000”原始 CSV 数据集的开头部分。

```
table with row headers in column A and column headers in rows 3 through 4,,,,,,,
,,,
"Table 1. Annual Estimates of the Resident Population for Incorporated Places
Over 50,000, Ranked by July 1, 2011 Population: April 1, 2010 to July 1, 2011"
,,,,,,,,,,
Rank,Geographic Area,,"April 1, 2010",,Population Estimate (as of July 1),,
,,,,Place,State,Census,Estimates Base,2010,2011,,,,
1,New York city,New York,"8,175,133","8,175,133","8,186,443","8,244,910",,,,
2,Los Angeles city,California,"3,792,621","3,792,625","3,795,761","3,819,702"
,,,,
3,Chicago city,Illinois,"2,695,598","2,695,598","2,698,283","2,707,12
0",,,,
4,Houston city,Texas,"2,099,451","2,099,430","2,108,278","2,145,146",,,,
5,Philadelphia city,Pennsylvania,"1,526,006","1,526,006","1,528,074","1,536,471"
,,,,
6,Phoenix city,Arizona,"1,445,632","1,445,656","1,448,531","1,469,471",,,,
7,San Antonio city,Texas,"1,327,407","1,327,606","1,334,431","1,359,758",,,,
8,San Diego city,California,"1,307,402","1,307,406","1,311,516","1,326,179",,,,
9,Dallas city,Texas,"1,197,816","1,197,816","1,201,715","1,223,229",,,,
10,San Jose city,California,"945,942","952,612","955,091","967,487",,,,
...
```

确实挺乱的，而且我们也不需要那么全面的数据。于是我在自己的电子表格程序里打开这个 CSV，然后做了一些清理工作，删除了不需要的数据列。（建议使用 LibreOffice Calc、Apple Numbers 或 Microsoft Excel）。而且，我们只想得到最大的 50 个城市的数据，所以就把其他城市的数据都删掉了。再导出为 CSV，就有了以下数据：

```
rank,place,population
1,New York city,8175133
2,Los Angeles city,3792621
3,Chicago city,2695598
4,Houston city,2099451
5,Philadelphia city,1526006
6,Phoenix city,1445632
7,San Antonio city,1327407
8,San Diego city,1307402
9,Dallas city,1197816
10,San Jose city,945942
...
```

这些都是有用信息，但要把它们标注到地图上，还需要每个城市的经纬度坐标信息。如果手工查找，那得花很长时间。所幸，我们可以利用一些现成的地理编码服务来提高效率。地理编码（geocoding）服务能够根据地名查找地图（实际上是查找数据库），返回精确的经纬度坐标。说“精确”可能有点夸大，虽然地理编码程序会尽力确保这一点，但在遇到有歧义的时候也免不了会返回错误的数据。比如，要查询Paris，返回的可能是法国巴黎，而不是德克萨斯州帕里斯的信息。因此，在根据地理编码服务返回的数据标注完城市后，最好用眼睛测试一下地图，再手工修改一遍那些错误的坐标。（以 http://teczno.com/squares 作为参考。）

在常用的地理编码应用里，我把这些地名粘贴进去，单击“开始”！几分钟后，就得到了结果，结果中包含更多逗号分隔的值，其中就有纬度 / 经度信息。把这些信息导入电子表格程序，再保存一份格式统一的带坐标的 CSV 文件：

```
rank,place,population,lat,lon
1,New York city,8175133,40.71455,-74.007124
2,Los Angeles city,3792621,34.05349,-118.245323
3,Chicago city,2695598,45.37399,-92.888759
4,Houston city,2099451,41.337462,-75.733627
5,Philadelphia city,1526006,37.15477,-94.486114
6,Phoenix city,1445632,32.46764,-85.000823
7,San Antonio city,1327407,37.706576,-122.440612
8,San Diego city,1307402,37.707815,-122.466624
9,Dallas city,1197816,40.636,-91.168309
10,San Jose city,945942,41.209716,-112.003047
...
```

整个过程比想象的要简单得多。如果是十年前，我们为此可能得花上好几个小时，一条一条地查询，然后再整理好数据。而现在呢，复制加粘贴，根本不用动什么脑子，几分钟就搞定了。相信读者知道为什么在线地图这么火爆了。

数据准备就绪，而且我们也知道怎么加载它们了：

```
d3.csv("us-cities.csv", function(data) {
        // 加载完数据，执行一些操作
});
```

在这个回调函数内部，可以用代码表达怎么用新创建的 circle 元素代表每个城市。然后，根据各自的地理坐标，将它们定位到地图上：

```
svg.selectAll("circle")
   .data(data)
   .enter()
   .append("circle")
   .attr("cx", function(d) {
           return projection([d.lon, d.lat])[0];
   })
```

```
        .attr("cy", function(d) {
                return projection([d.lon, d.lat])[1];
        })
        .attr("r", 5)
        .style("fill", "yellow")
        .style("opacity", 0.75);
```

以上代码的关键是通过 `attr()` 语句设定 `cx` 和 `cy` 值。没错，可以通过 `d.lon` 和 `d.lat` 获得原始的经度和纬度。但我们真正需要的，则是在屏幕上定位这些圆形的 *x*/*y* 坐标，而不是地理坐标。

因此就要借助 `projection()`，它本质上是一个二维比例尺方法。给 D3 的比例尺传入一个值，它会返回另一个值。对于投影而言，我们传入两个数值，返回两个数值。（投影和简单的比例尺的另一个主要区别是后台计算，前者要复杂得多，后者只是一个简单的归一化映射。）

地图投影接受一个包含两个值的数组作为输入，经度在前，纬度在后（这是 GeoJSON 格式规定的）。然后，投影就会返回一个包含两个值的数组，分别是屏幕上的 *x*/*y* 坐标值。因此，设定 `cx` 时使用 `[0]` 取得第一个值，也就是 *x* 坐标值；设定 `cy` 时使用 `[1]` 取得第二个值，也就是 *y* 坐标值。明白了吗？

结果就是图 12-6 所示的带城市的地图，太好了！代码请参考 06_points.html。

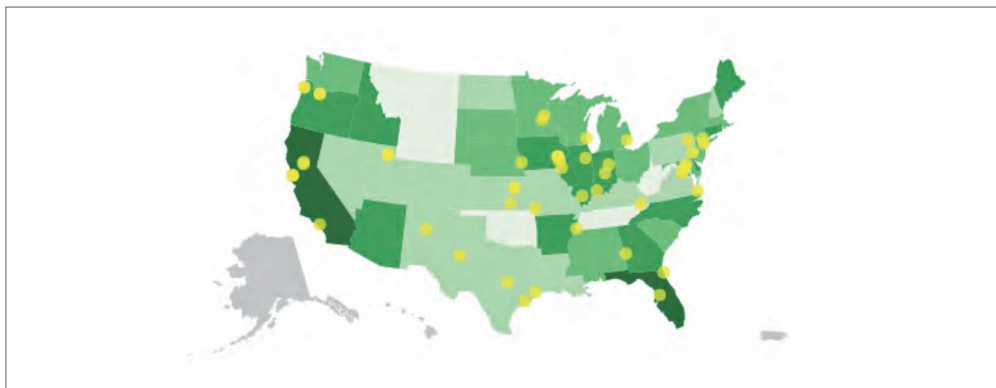


图 12-6：美国最大的 50 个城市，在地图上用可爱的小黄点表示

不过，这些点大小都一样啊，应该把人口数据反映到圆形大小上。为此，可以这样引用人口数据：

```
        .attr("r", function(d) {
              return Math.sqrt(parseInt(d.population) * 0.00004);
        })
```

这里先取得 `d.population`，把它传给 `parseInt()` 实现从字符串到整数值的转换，再随便乘一个小数降低其量级，最后得到其平方根（把面积转换为半径）。代码参见 07_points_sized.html。

如图 12-7 所示，最大的城市突出出来了。城市大小的差异很明显，这种情况可能更适合使用对数比例尺，尤其是在包含人口更少的城市的情况下。这时候就不用乘以 0.00004 了，可以直接使用自定义的 D3 比例尺函数。（作为一个练习留给大家吧。）

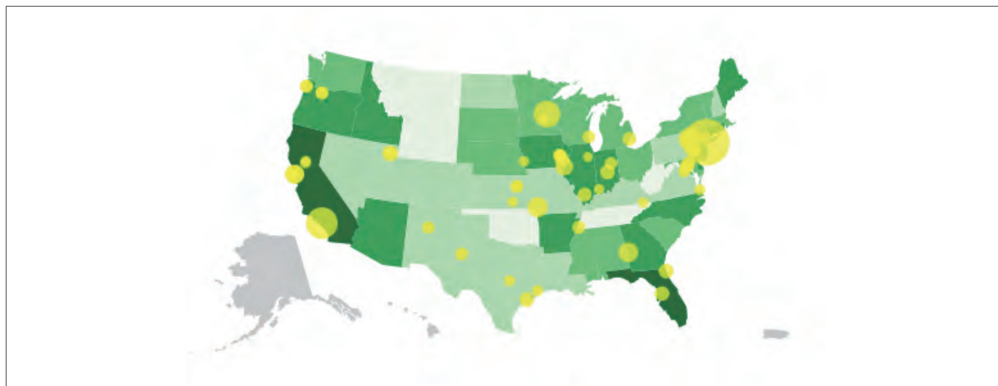


图 12-7：表示城市的圆点面积对应着人口

这个例子的关键在于，我们整合了两个不同的数据集，把它们加载并显示在了地图上。（如果算上地理编码坐标，一共就是三个数据集了！）

12.6 取得和解析地图数据

如果只想生成美国的地图，那我们手里已经有很多 GeoJSON 数据了。但世界之大，可能还有很多地区需要我们去为它生成地图。本节我们就专门讲一讲怎么去查找和解析其他地区的地图数据。最终目标是能生成像 us-states.json 一样的 GeoJSON 数据，以便在 D3 中使用。

12.6.1 查找shapefile文件

所谓的 shapefile，是在线地图和可视化流行之前的一种文件。这种文件包含着与现在的 GeoJSON 中几乎相同的数据，比如地理区域的边界、这些区域中的点等等。只不过格式不是纯文本，因此很难用代码来读取。shapefile 是使用 GIS（Geographic Information Systems，地理信息系统）软件的地理工作者、测绘人员和科研人员常用的一种格式。如果你能接触到昂贵的 GIS 软件，那么对 shapefile 应该不会陌生。但这种人毕竟只是少数，浏览器不可能去兼容这种格式。

如果你实在找不到合用的 GeoJSON 文件，可以试着找一找 shapefile 文件。政府网站一般是首选目标，特别是某些国家或地区的政府网站。以下是我最常用的两个资源。

- Natural Earth（http://www.naturalearthdata.com/）
  集合了公共域中的大量地理数据，包括文化、政治和自然特征信息。绘制国家地图涉及政治敏感问题，Natural Earth 详细说明了它们的设计原则。

- 美国人口普查局（http://www.census.gov/geo/www/cob/cbf_state.html）
  可以找到美国每个州的边界数据，还包含县、公路、水文等特征信息，非常丰富，都在公共域。

12.6.2 选择解析度

下载之前，要确定数据的解析度（resolution）。所有 shapefile 都是矢量数据（而不是位图），因此解析度的意思不是像素，而是地理信息的详尽程度或粒度。

Natural Earth 的数据集分三种解析度，包括最详尽的和不怎么详尽的：

- 1:10 000 000
- 1:50 000 000
- 1:110 000 000

也就是说，在最高解析度的数据中，1 个单位对应现实世界中的 1000 万个这样的单位。或者反过来说，现实世界中的 1000 万个单位会简化为 1 个。因此，1000 万英寸（254 公里）在这个数据中表示为 1 英寸。

这种解析度可以表达为更简单的形式：

- 1:1000 万
- 1:5000 万
- 1:11 000 万

对于不太详尽（“缩小后”）的地图，1:11 000 万的解析度是可以胜任的。如果想显示每个州的详细轮廓，则 1:1000 万效果更好。如果想在地图上显示一个非常小（“放大后”）的区域，例如特定的城市甚至街区，那就得去找更高解析度的数据。（看看本州或本市政府的网站。）

不同的数据源提供的数据解析度也可能不同。美国人口普查局的很多 shapefile 文件对应于下列三种比例尺：

- 1:500 000（1:50 万）
- 1:5 000 000（1:500 万）
- 1:20 000 000（1:2000 万）

选好解析度后下载文件，得到的一般是 ZIP 压缩文件，包含其他一些文件。比如，我从 Natural Earth 上可以下载一个 1:11 000 万（低解析度）的海洋文件：http://www.naturalearthdata.com/downloads/110m-physical-vectors/110m-ocean/。

解压缩后，可以得到如下几个文件：

```
ne_110m_ocean.dbf
ne_110m_ocean.prj
ne_110m_ocean.README.html
ne_110m_ocean.shp
ne_110m_ocean.shx
ne_110m_ocean.VERSION.txt
```

这些扩展名乱七八糟的都是什么啊?！别急，我们只关心以 .shp（shapefile）结尾的那个文件，但其他文件暂时也不用删掉。

12.6.3 简化数据文件

理想情况下，你可能会找到解析度正合适的 shapefile 文件。可是，如果你只能找到超高解析度的文件，比如 1:10 万的，怎么办呢？这种文件可能会非常大。而作为 JavaScript 程序员，你可能又非常在乎效率，还记得吗？不要向浏览器发送几兆的地理数据。

好在，我们可以简化这些文件，也就是把它们转换成低解析度的版本。简化数据的过程在 Mike Bostock 的这篇文章里，被非常唯美地展示了出来：http://bost.ocks.org/mike/simplify/。

Matt Bloch 开发的 MapShaper（http://mapshaper.org/）是一个非常好用的简化数据的工具。你可以上传 shapefile 文件，然后拖动滑动条上的滑块来选择解析度。

使用 MapShaper 时，以“Shapefile – polygons”作为导出选项。这样可以同时生成一个 .shp 文件和一个 .shx 文件。下载这两个文件，把它们重命名为与原来的 .shp 和 .shx 文件一致的名字。然后在把它们转换到 GeoJSON 格式前，备份一下原来的 .dbf 文件，并将它与简化后的 shapefile 文件放在同一个文件夹里。这一步很重要，可以保证不丢失保存在 .dbf 中的元数据信息，比如县 ID 或其他路径标识符。

另外，还可以考虑使用 Mike Migurski 的 Bloch（https://github.com/migurski/Bloch），这是对 Matt Bloch 简化算法的一个 Python 实现，或者 d3.simplify 插件（Mike 前

面的演示示例中使用的就是这个插件)。但愿有一天可以使用 JavaScript 直接实现简化,然后导出为可以在项目中使用的 JSON 格式。这类工具发展很快,因此请大家平时多关注一下!(实际上,在我写到这一段时,Bostock 就发布了一个用于几何简化的新项目 TopoJSON(https://github.com/mbostock/topojson)的演示(http://bl.ocks.org/4090870)。你说这个世界变化有多快吧!恐怕在你看到这一段文字时,这个 TopoJSON 命令行工具已经可以成为你的得力工具了。这个工具的功能包括加载 shapefile 文件、执行简化并将数据转换为 JSON 格式。跟我们期待的一样,TopoJSON 考虑到了 D3,虽然它输出新的 TopoJSON 格式,但这种格式与 GeoJSON 类似,而且效率更高。)

12.6.4 转换为GeoJSON

如果没有合适的软件,那么这一步会麻烦一些。我们最终的目的是要使用一个叫 `ogr2ogr` 的终端命令,可以在 Mac、Unix 和 Windows 系统中使用。主要问题是 `ogr2ogr` 依赖一些框架、库之类的东西,不把它们都安装好,就没办法用。

关于安装这些依赖的细节,我们就不介绍了,但本节会告诉你大致怎么做。

首先,要下载 Geospatial Data Abstraction Library,即 GDAL(http://www.gdal.org/)。这个程序包里包含 `ogr2ogr`。

还要下载 GEOS(http://trac.osgeo.org/geos/),即 Geometry Engine, Open Source。

然后在 Windows 或 Unix/Linux 计算机中,下载源代码,然后输入好玩的 `build`、`make`,以及众多其他安装命令。

实际的安装命令我不记得了,但大致过程就是这样。(郑重地跟大家说一声,如果这一步你过不了,可以参考 O'Reilly 出版的相关图书,根据里面的介绍下载和安装这种软件包。)

如果你使用的是 Mac,那很可能已经安装了 Xcode 和 Homebrew。那么,只要在终端中简单地输入 `brew install gdal`,就行了。(如果这两个工具有一个你没安装,建议安装上。这两个工具都是免费的,但安装可能得花点时间。Xcode 本身很大,要从 App Store 下载。安装了 Xcode 之后,至少从理论讲,只要在终端里使用一个简单的命令就能安装 Homebrew 了。根据我的经验,可能要解决一些小问题才能最终安装好。)

对于使用 Mac 但没有安装 Xcode 或 Homebrew 用户来说,还可以选择一个预编译的 GUI 安装程序,这个程序会安装 GDAL、GEOS 以及其他一些你不用知道是干

什么的工具。GDAL Complete 包的最新版地址在这里：http://www.kyngchaos.com/software/frameworks。仔细看一看 GDAL ReadMe 文件吧。安装后，还不能在终端窗口里使用 ogr2ogr。还得把 GDAL 程序放到壳程序的路径中。最简单的办法是打开终端窗口，输入 nano .bash_profile，然后把 export PATH=/Library/Frameworks/GDAL.framework/Programs:$PAT 粘贴进去，再按 Control-X 和 Control-y 保存，再输入 exit 退出会话。打开一个新终端窗口，输入 ogr2ogr 就能看到它可以使用了。

无论你使用什么操作系统，安装了这些工具后，打开终端窗口，进入保存所有 shapefile 文件的文件夹（比如 cd ~/ocean_shapes/），然后输入如下命令：

```
ogr2ogr -f "GeoJSON" output.json filename.shp
```

这是告诉 ogr2ogr 取得扩展名为 .shp 的 filename 文件，把它转换成 GeoJSON，然后保存为名为 output.json 的文件。

以我下载的海洋文件为例，使用 ogr2ogr 的命令如下：

```
ogr2ogr -f "GeoJSON" output.json ne_110m_ocean.shp
```

输入这些命令，但愿你什么也看不到。

这就完了啊？！我知道，花几个小时间才弄好了命令行工具，你希望结尾怎么也得辉煌一些吧，就像你在《超级马里奥兄弟 3》里救下公主一样。（其实我始终没玩到那一关，但我想象着那一刻肯定非常激动人心。）

可是没有，你最好乞求什么也不会发生。当然，同一个文件夹里最好还会多一个叫 output.json 的文件。这就是我得到的结果：

```
{
"type": "FeatureCollection",
"features": [ { "type": "Feature", "properties":
{ "scalerank": 0, "featurecla": "Ocean" },
"geometry": { "type": "Polygon", "coordinates":
    [ [ [ 49.110290527343778, 41.28228759765625 ],
        [ 48.584472656250085, 41.80889892578125 ],
    [ 47.492492675781335, 42.9866943359375 ],
        [ 47.590881347656278, 43.660278320312528 ],
    [ 46.682128906250028, 44.609313964843807 ],
        [ 47.675903320312585, 45.641479492187557 ],
    [ 48.645507812500085, 45.806274414062557 ]
        ...
```

嘿，最后这个结果看起来很面熟啊！

现在，你就可以高高兴兴地把新生成的 GeoJSON 复制到 D3 文件夹里了。我把它重

命名为 oceans.json，复制了之前的一个 HTML 文档，然后在 D3 代码里简单地把对 us-states.json 的引用改为引用 oceans.json，就得到了图 12-8 所示的结果。



图 12-8：GeoJSON 数据可视化，呃，这是地球上的海洋？

老兄，这是什么东西?！不管是什么，你先看看 08_oceans.html。

匆忙之间，我忘了更新投影了！再改动一小点，把 `albersUsa` 改成 `mercator`（参见图 12-9）。



图 12-9：GeoJSON 数据可视化，这才是投影正确的全球海域图嘛

请大家参考 09_mercator.html，其中包含了海洋的 GeoJSON 路径，以及下载、解析和可视化的所有代码。

第13章

导出文件

有时候需要把图表用在浏览器之外的环境下，比如你接到 TED 的邀请去做一次演讲，或者在 MoMA（纽约现代艺术博物馆）办自己的第一次个展。

本章介绍三种把 D3 生成的图表导出为其他格式的简单方法。D3 并没有内置的“导出”函数（有些人自己写过），因此下面只是几种适用于任何在浏览器中生成的 SVG 图形的技术。

13.1 导出位图

导出位图最简单的方法就是截屏，当然图片品质也最低。根据你使用的操作系统，可以在 PC 中按键盘上的 Print Screen 键，或者在 Mac 上按⌘-Shift-4（拖动十字光标选择要截取的区域，松开鼠标，就可以在桌面上看到生成的 PNG 图片）。

图 13-1 就是我用上述方法截取的位图图片。



图 13-1：一张 PNG 截图

这种方法简单快捷，但位图图片的分辨率只有屏幕的分辨率那么大。因此，这张图片不可能放得太大，打印出来也不会很清晰。这种低分辨率图片一般只适合屏幕显示。（当然，如果你有一台像素密度超高的显示器，那另当别论。）

13.2　导出PDF

PDF，即 Portable Document Format（便携文档格式）的文档可以包含矢量图，包括 SVG 图形。因此，导出到 PDF 可以迅速得到一个可伸缩的图表（参见图 13-2）。



图 13-2：PDF 可以保持原始的矢量数据，因而很清晰

在 Mac 上打开浏览器，选择“文件 > 打印”。然后找到 PDF 菜单，选择“保存为 PDF”。在 Windows 或 Linux 上，可能需要安装第三方插件才能有打印到 PDF 的功能。（我听说 doPDF for Windows 可以使用，而且免费。）

13.3　导出SVG

既然我们使用 D3 生成了 SVG 格式的图形，那为什么不把它直接保存成 SVG 格式呢？这样就跟导出 PDF 格式没有什么差别了，既可以保持原始的矢量数据，从而确保可以伸缩，又能把导出的文件导入 Illustrator 或其他 SVG 编辑器，然后进行调整。（在某种程度上，这对于 PDF 文件也是可行的，但要看使用的是什么 PDF 生成器，有些生成器生成的 PDF 会以意想不到的方式对元素进行分组和分层。）

最简单的办法就是从 DOM 中直接复制 SVG 代码（参见图 13-3）。首先，检查 SVG 元素，在 Web 检查器中点击该元素，然后选择“复制”或“复制为 HTML”。

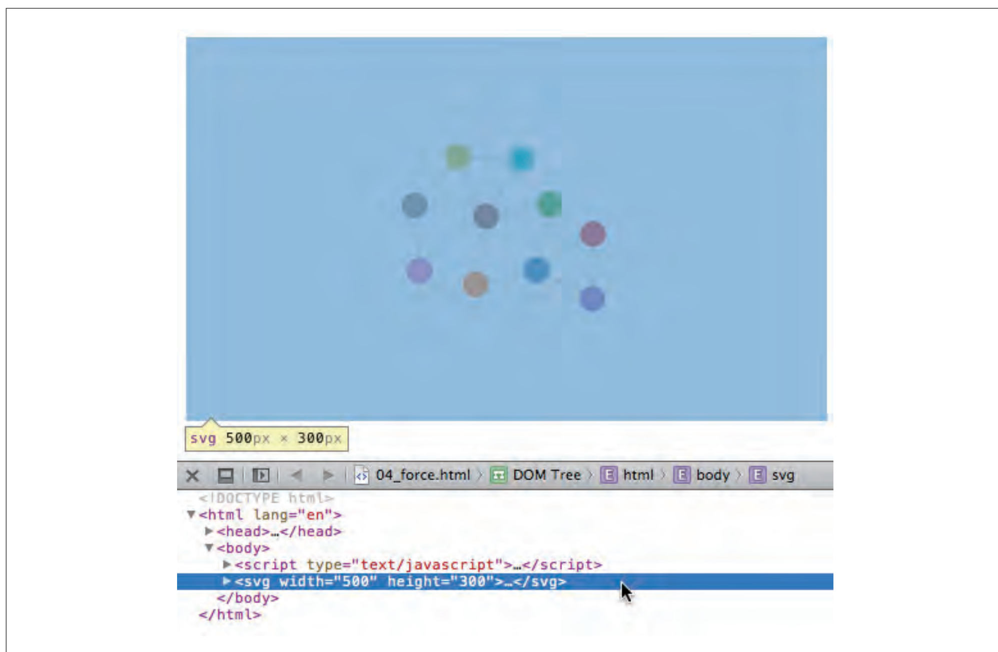


图 13-3：从 DOM 中复制 D3 生成的 SVG 代码

切换到文本编辑器，把复制的代码粘贴到一个新文件中，如图 13-4 所示。

然后将新文件保存为 something.svg。用 Illustrator（或其他 SVG 编辑器）打开这个文件，如图 13-5 所示。

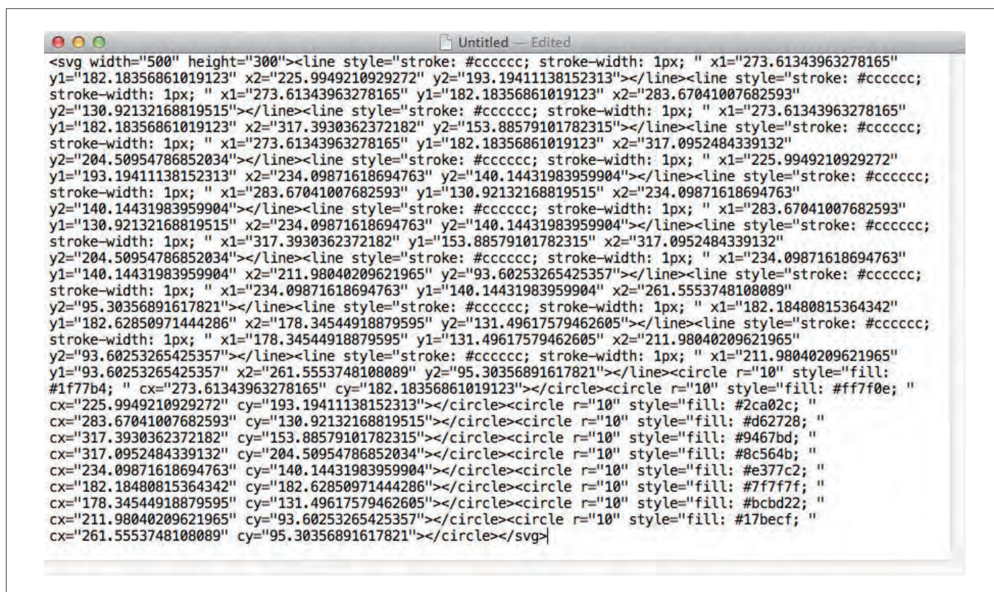


图 13-4：把 SVG 代码粘贴到新文件中

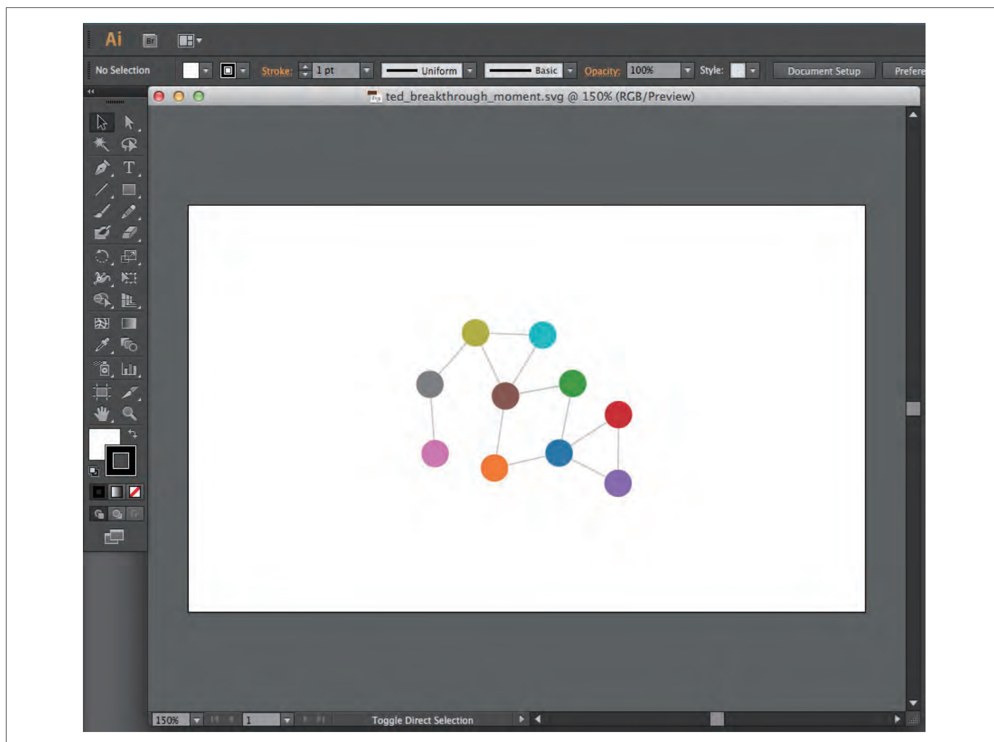


图 13-5：在 Illustrator 中打开导出的 SVG 文件

如图 13-6 所示，所有元素可都可以单独选中并编辑。在图 13-7 所示的版本中，我采用图弗特（Tufte）增强技术，添加了可变权重的线条，而且为了让人心态平和地看这张图，还添加了中性的淡紫色。（声明一下，我是开玩笑的，我和图弗特都不同意这种做法。）

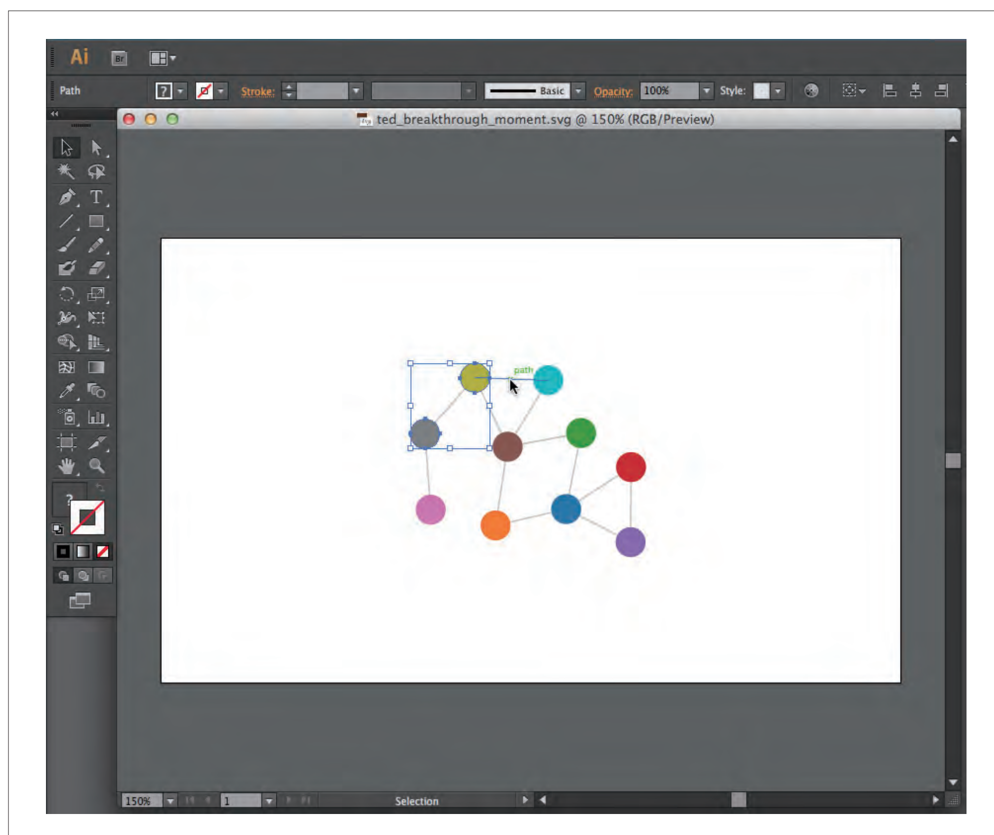


图 13-6：选择了一个 SVG 元素

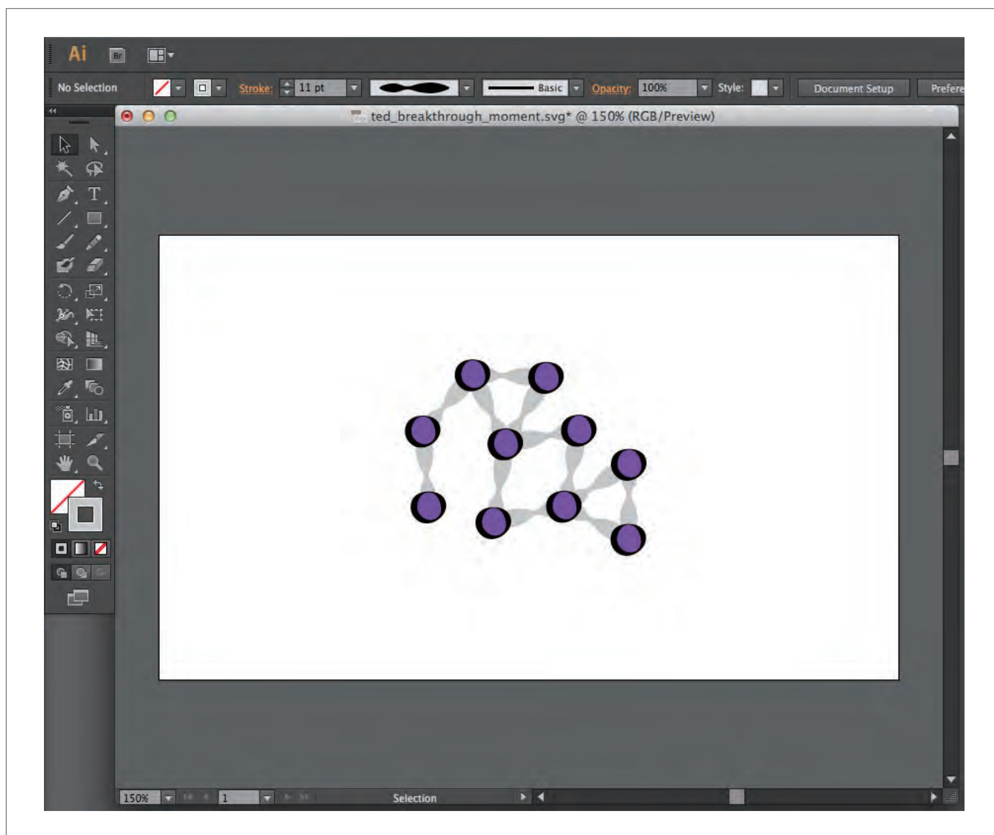


图 3-17：我的开创性设计，通过手工调整，开创性更加鲜明

好了，把 SVG 图形导出到浏览器之外的几种方式讲完了。可以把导出的图形用于文档、演示、打印或屏幕显示。

附录A

扩展阅读

现在你是个有经验的人啦。本书介绍了很多关于 D3 的基本概念，而你对它的基本用法和常见技术也有了较好的理解。如果说你已经学到了什么，那我希望你知道任何一个任务都有几种（或者几十种，几百种）解决方案，而这正是编程的乐趣所在，对吧？本书展示的方式，都是最简单和最直观的方式，而且理解起来也最容易，至少对我而言是这样。可是，也许还有其他更好方法来解决本书中提出的各种问题。所谓“更好”，意思是“计算效率更高”或“最适合你或你的工作方式”。我倾向于后一种定义。编程就像解谜题，你必须自己想办法把计算机该干什么怎么干告诉它——用一种你作为人类同样能理解的语言。

D3 是一个强大的工具，我们只是对它有了一个大致的了解。只有真正做几个自己的可视化项目，才会发现其他有用的方法和各种前所未知的快捷方式。本书还没有讲到的东西很多，比如 D3 用于处理日期和时间值、动态选择和计算颜色、快捷操作数组，以及客户端数据处理必需的各种内置方法。关于 D3，要学的东西还有很多。但我已经把最核心的概念讲完了，剩下的就得靠你自己去探索了。

好，那接下来怎么探索？本附录给出了一些有用的资源，或许可以满足你的需求。不要忘了，D3 本身仍在不断改进，而这些资源也一样。在你阅读到这里的时候，一些这里没有提到的新网站或教程也许又出现了。正因为如此，我也不推荐你只是一味地看教程，而是要深入 D3 的社区。加入它的 Google Group，在 Twitter 上关注一批人，时刻注意最新动向。有问题就找人讨论讨论，结交几位数据可视化方面的同好。如果当地还没有 D3 的数据可视化小组，你自己创办一个。总之，只有多向别人学习，自己才能快速提高。

不管怎么说，学习完这本书，（不管愿不愿意）你已经算是一位 D3 社区的成员啦，欢迎！

A.1　图书

Getting Starting with D3，作者 Mike Dewar（O'Reilly，2012）

是的，在本书写作时，市面上只有一本关于 D3 的书。Mike Dewar 这本书涵盖了一些高级主题，也包含基于纽约市交通系统的一些真实数据的示例。（有意思！）

A.2　网站

- d3js.org
  学习 D3 一切的起点。
- github.com/mbostock/d3/wiki/Gallery
  D3 的代码库，包含数以百计的示例。把你作品也加进去！
- bl.ocks.org/mbostock
  例子更多，而且都是 Mike Bostock 亲手编写的，每一个都反映了 D3 的一项功能。
- github.com/mbostock/d3/wiki/API-Reference
  D3 API 参考，包括所有方法和参数的详细说明。
- stackoverflow.com/questions/tagged/d3.js
  遇到什么难题的时候，可以在 StackOverflow 上用 d3.js 标签提个问。
- groups.google.com/forum/?fromgroups#!forum/d3-js
  鱼龙混杂的 D3 Google Group。在这里可以看到最新的项目和进展。（技术问题一定到 StackOverflow 上问。）
- bl.ocks.org
  用于发表托管在 GitHub Gist 中的代码，作者是 Mike Bostock。非常适合快速与他人分享你的代码，比如在 StackOverflow 上寻求帮助的时候，或者显摆你在 Google Group 上挖到的新玩艺儿的时候，都可以用到它。
- blog.visual.ly/creating-animations-and-transitions-with-d3-js/
  Jérôme Cukier 写的一个关于使用 D3 来创建动画和过渡效果的教程，极其精彩，有很多当场可交互的例子。
- d3noob.org
  新的介绍 D3 技巧和经验的资源网站，不错。

- tributary.io
  用于试验 D3 代码的一个实时编码环境，作者是 Ian Johnson。

- D3 Plug-ins（https://github.com/d3/d3-plugins）
  包含扩展 D3 功能的所有官方插件，以备不时之需。

A.3 Twitter

Twitter 是发现新项目和 D3 进展情况的好地方。下面给出一些值得关注的人，但肯定会遗漏一些同样值得关注的账号。

- @mbostock
  Mike Bostock，了解 D3 的进展和《纽约时报》新的可视化项目。

- @jasondavies
  Jason Davies，了解地理和数学映射的各种实验方法。

- @d3visualization
  Christophe Viau，了解 D3 世界的各种更新。

- @enjalot
  Ian Johnson，了解新编码工作和技术，还有视频教程。

- @syntagmatic
  Kai Chang，了解非常多的 D3 设计方法，难以计数。

- @jcukier
  Jérôme Cukier，了解极具创造性的可视化项目，还有关于流程每一步的关键提示。

- @darkgreener
  Anna Powell-Smith，了解探索个人数据的交互可视化项目，非常漂亮。

- @vlandham
  Jim Vallandingham，了解做项目过程中的注意事项，还有不错的教程。

- @alignedleft
  Scott Murray，了解本书勘误、更新和修订，以及未来的数据驱动的项目。

我觉得大多数该关注的男同胞都在这里头了。在写作本书时，D3 用户和贡献者社区的主要成员好像都是男性。希望将来这本书出新版的时候，这个列表可以更丰富多彩一些。

作者简介

Scott Murray 是一位编码艺术家，他的工作是编写代码来创建可视化的数据图表及其他交互式作品。他的作品涉及交互设计、系统设计和生成艺术。

Scott 是旧金山大学助理教授，主要讲授数据可视化和交互设计。他是 Processing（processing.org）的贡献者，也在培训班上讲创造性编码。

Scott 拥有瓦萨学院的文学学士学位、马萨诸塞州艺术与设计学院动态媒体研究所美术硕士学位。他的个人作品展示站点是 alignedleft.com。

封面说明

本书封面上的动物是长尾山雀，也叫丛山雀（银喉长尾山雀）。丛山雀是欧洲和亚洲各地常见的一种鸟类，其中长尾的山雀头部为纯白色。

这种鸟以其体型小巧而闻名，算上尾巴也只有大约 13~15 厘米长。长尾山雀的喙短而粗，与其修长的尾巴相映成趣。雌性和雄性长尾山雀都会在出生后第一个冬天前全身脱一次毛，性别难辨。成年山雀的羽毛主要是黑色和白色，间杂一些灰色和粉色。

丛山雀栖息在落叶林和混合林中，喜食虫卵以及飞蛾和蝴蝶的幼虫，喜欢橡树、白蜡树以及无花果树。一般把巢筑在距离地面较近的灌木丛。

数据可视化实战：使用D3设计交互式图表

你手头有一些数据，想做成漂亮的图表放到网站上？好主意，通过浏览器来跨平台实现数据可视化是正确的选择。什么，你还想让图表能够响应用户操作？没问题，交互式图表比静态图片更能吸引人去探究本源。好啦，要生成通过浏览器展示的动态图表，首选目前最热门的Web数据可视化库——D3。

这本书很有意思，而且对读者要求不高。不需要知道什么是数据可视化，也不用有太多Web开发背景就能看懂它。不信？翻一翻就知道这是一本既好玩又实用的动手指南啦！看完这本书你会怎么样呢？

- 掌握必要的HTML、CSS、JavaScript和SVG基础知识；
- 学会基于数据在网页里生成元素和为它们设置样式的技巧；
- 能够生成条形图、散点图、饼图、堆叠条形图和力导向图；
- 使用平滑的过渡动画来展示数据的变化；
- 赋予图表动态交互能力，响应用户从不同角度探索数据的请求；
- 收集数据和创建自定义的地图；
- 另外，本书100多个代码示例都可以在线浏览！

“难懂的技术细节到了作者Scott Murray的笔下，三言两语就讲得清清楚楚。假如你早就想探索基于Web标准来实现动态的数据可视化——就算没多少编程经验，这本书都是你最合适的选择！”

——Mike Bostock
最有潜力的Web数据可视化库D3的创造者

Scott Murray

编码艺术家，旧金山大学助理教授，主要讲授数据可视化和交互设计。他是Processing（processing.org）的贡献者，个人作品站点是alignedleft.com。

封面设计：Karen Montgomery 张健

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/网页设计/数据展示

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

O'REILLY®
oreilly.com.cn





ISBN 978-7-115-32011-7
定价：59.00元

欢迎加入

图灵社区

电子书发售平台

电子出版的时代已经来临，在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版的梦想。你可以联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者，这极大地降低了出版的门槛。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果有意翻译哪本图书，欢迎来社区申请。只要通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

读者交流平台

在图灵社区，读者可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。欢迎大家积极参与社区开展的访谈、审读、评选等多种活动，赢取银子，可以换书哦！